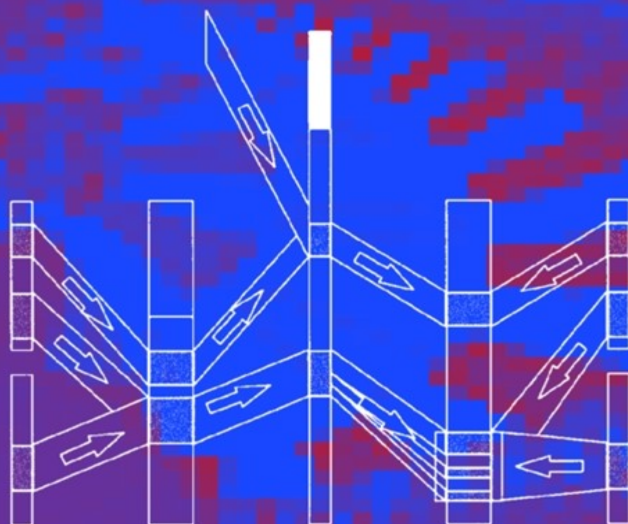
LNCS State-of-the-Art Survey

Hermann Hellwagner
Alexander Reinefeld (Eds.)

SCI: Scalable Coherent Interface

Architecture and Software for High-Performance Compute Clusters



Springer

Lecture Notes in Computer Science 1734
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Hermann Hellwagner
Alexander Reinefeld (Eds.)

SCI: Scalable Coherent Interface

Architecture and Software
for High-Performance Compute Clusters

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Hermann Hellwagner
University of Klagenfurt, Institute of Information Technology
A-9020 Klagenfurt, Austria
E-mail: hermann.hellwagner@uni-klu.ac.at

Alexander Reinefeld
Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
Takustr. 7, D-14195 Berlin-Dahlem, Germany
E-mail: ar@zib.de

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

SCI - Scalable coherent interface : architecture and software for high-performance compute clusters / Hermann Hellwagner ; Alexander Reinefeld (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1734)
ISBN 3-540-66696-6

CR Subject Classification (1998): C.2, D.1-4, B.2-8

ISSN 0302-9743
ISBN 3-540-66696-6 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1999
Printed in Germany

Typesetting: Camera-ready by author
SPIN: 10704208 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

Background

System interconnection networks have become a critical component of the computing technology of the late 1990s, and they are likely to have a great impact on the design, architecture, and use of future high-performance computers. Indeed, it is today not only the sheer computational speed that distinguishes high-performance computers from desktop systems, but the efficient integration of the computing nodes into tightly coupled multiprocessor systems. Network adapters, switches, and device driver software are increasingly becoming performance-critical components in modern supercomputers.

Due to the recent availability of fast commodity network adapter cards and switches, tightly integrated clusters of PCs or workstations have emerged on the market, now filling the gap between desktop systems and supercomputers. The use of commercial off-the-shelf (COTS) technology for both computing and networking enables scalable computing at relatively low costs. Some may disagree, but even the world champion in high-performance computing, Sandia Lab's *ASCI Red* machine, may be seen as a COTS system. With just one hardware upgrade (pertaining to the Intel processors, not the network), this system has constantly been number one in the TOP-500 list of the worldwide fastest supercomputers since its installation in 1997. Clearly, the system area network plays a decisive role in overall performance.

The Scalable Coherent Interface (SCI, ANSI/IEEE Standard 1596-1992) specifies one such fast system interconnect, emphasizing the flexibility, scalability, and high performance of the network. In recent years, SCI has become an innovative and widely discussed approach to interconnecting multiple processing nodes in various ways. SCI's flexibility stems mainly from its communication protocols: in contrast to many other interconnects, SCI is not restricted to either message-based or shared-memory communication models. Instead, it combines both, taking advantage of similar properties that have been investigated in such hybrid machines as Stanford's FLASH or MIT's Alewife architectures. Since SCI also defines a distributed directory-based cache coherence protocol, it is up to the computer architect to choose from a broad range of communication and execution models, including efficient message-passing architectures, as well as shared-memory models, in either the NUMA or CC-NUMA variants.

European industry and research institutions have played a key role in the SCI standardization process. Based on SCI adapter cards, switches, and fully integrated cluster systems manufactured by European companies, the SCI community in Europe has made and is making significant developments and state-of-the-art research on this important interconnect.

Purpose of the Book

From many discussions with friends, colleagues, and potential users, we found that one significant barrier to the widespread deployment and use of SCI is the lack of a clear vision of how SCI works, how it is being used in building clusters, and how obstacles in its deployment can be avoided. Our goal in compiling this book is to address these barriers by providing in-depth information on the technology and applications of SCI from various perspectives. The book focuses on SCI clusters built from commodity PCs or workstations and SCI adapters, since they represent the mainstream and most cost-effective application of SCI to date.

In addition, some challenging research issues, mostly pertaining to shared-memory programming on SCI clusters, are discussed and potential improvements for SCI cluster equipment are highlighted.

Who is the intended audience? The relevance of the book for computer architects is obvious, given the importance of system area networks for modern high-performance computers. But the book is also intended for system administrators and compute center managers who plan to invest in cluster technology with COTS components. Furthermore, researchers and students wanting to contribute to this interesting technology with their own hard- or software developments might find this book helpful.

Organization of the Book

The book consists of nine parts, each subdivided into chapters covering individual topics. On the whole, the contributions cover the complete hardware/software spectrum of SCI clusters, ranging from the major concepts of SCI, through SCI hardware, networking, and low-level software issues, various programming models and environments, up to tools and application experiences.

Part I introduces the SCI standard and its application in practical computer systems. SCI is put into context by comparing its concepts, architecture, and performance with its strongest competitor Myrinet and also with the proprietary Cray T3D interconnection network which set the standards back in 1993.

Part II looks at the hardware. It describes two implementations of SCI adapters, the commercial, widely used Dolphin SCI cards for the PCI and SBus I/O buses, and the prototype adapter developed at TU München which can be extended by special hardware for monitoring the SCI packet flow.

Building on the hardware, Part III explores how to build SCI interconnection networks and analyzes various critical aspects of SCI networks, among them ringlet scalability and potential performance degradation by hardware-generated retry traffic.

Part IV moves on to software, describing the functionality and concrete implementations of SCI device drivers and introducing a low-level API that abstracts away SCI's distributed shared memory (DSM) implementation details from higher-level software.

The first class of parallel and distributed programming models, namely message-passing libraries on top of SCI, are covered in Part V. The chapters report on projects which implemented sockets, TCP/IP, PVM, and MPI with high efficiency on top of SCI, by making judicious use of the SCI DSM and related features.

As pointed out by the contributions in Part VI, developing shared-memory programming environments on SCI clusters with current SCI hardware and driver software is more challenging than implementing message-passing libraries. Partly due to the lack of well established shared-memory standards, the approaches described are widely diverse. They range from specific shared virtual memory systems on top of SCI to a fully transparent, distributed thread system and to shared, parallel objects extending a CORBA middleware implementation. The chapters discuss some of the limitations of current SCI cluster equipment and present potential routes for future developments.

Real-world experiences with SCI clusters are reported in Part VII. As a reference, benchmark and application performance results from the very large SCI clusters that are operated at PC^2 Paderborn are given first. The parallelization approaches and performance results from two projects, a complex molecular dynamics code and a real-time data acquisition and filtering application prototype for high-energy physics, are described as examples of real-world uses of SCI clusters.

Part VIII deals with tools for SCI clusters, which apparently are still in their infancy. Therefore, only two basic SCI monitors, one implemented in hardware, the other in software, and their potential applications are presented here. In addition, a powerful system management tool, developed to operate the large Paderborn clusters as general-purpose, multi-user compute servers is introduced.

Both SCI and SCI interconnects are still evolving in terms of standardization, product development, research findings, and applications. In the final part, Part IX, therefore, one of the designers of SCI, David Gustavson, describes the perspectives that he sees for SCI.

Acknowledgements

With great pleasure, we acknowledge the efforts of the many individuals who have contributed to the development of this book. First and foremost, we thank the authors for their enthusiasm, time, and expertise which made this

book possible. We are also grateful to the people who helped in organizing the book, especially Oliver Heinz (PC^2 Paderborn), Hans-Hermann Frese (ZIB Berlin), and Angelika Rossak (University Klagenfurt). The European Commission provided financial support through the ESPRIT IV Programme's SCI Working Group (EP 22582). Finally, we acknowledge the help of Alfred Hofmann and Antje Endemann of Springer-Verlag, who were always competent, professional, and efficient partners to work with.

September 1999

Hermann Hellwagner
Alexander Reinefeld

Table of Contents

Part I. SCI and Competitive Interconnects for Cluster Computing

1. The SCI Standard and Applications of SCI
Hermann Hellwagner . . . 3

1.1 Introduction . . . 3
1.2 SCI Overview . . . 4
1.2.1 Background . . . 4
1.2.2 Goals . . . 4
1.2.3 Concepts . . . 6
1.2.4 Discussion . . . 11
1.3 The SCI Standard and Some Extensions . . . 11
1.3.1 Logical Layer . . . 12
1.3.2 Cache Coherence Layer . . . 19
1.3.3 Extensions . . . 22
1.4 Applications of SCI . . . 23
1.4.1 System Area Network for Clusters . . . 23
1.4.2 Memory Interconnect for Cache-Coherent Multiprocessors . . . 26
1.4.3 I/O Subsystem Interconnect . . . 30
1.4.4 Large-Scale Data Acquisition System . . . 31
1.5 Related Communication Networks and Concepts . . . 31
1.6 Concluding Remarks . . . 34

2. A Comparison of Three Gigabit Technologies: SCI, Myrinet and SGI/Cray T3D
Christian Kurmann, Thomas Stricker . . . 39

2.1 Introduction . . . 39
2.2 Levels of Comparison . . . 40
2.2.1 Direct Deposit . . . 41
2.2.2 Message Passing (MPI/PVM) . . . 42
2.2.3 Protocol Emulation (TCP/IP) . . . 44
2.3 Gigabit Network Technologies . . . 45
2.3.1 The Intel 80686 Hardware Platform . . . 46
2.3.2 Myricom Myrinet Technology . . . 47

2.3.3 Dolphin PCI-SCI Technology 48
2.3.4 The SGI/Cray T3D – A Reference Point 48
2.3.5 ATM: QoS – But Still Short of a Gigabit/s 50
2.3.6 Gigabit Ethernet – An Outlook 50
2.4 Transfer Modes 51
2.4.1 Overview 51
2.4.2 "Native" and "Alternate" Transfer Modes in the Three Architectures 54
2.5 Performance Evaluation 56
2.5.1 Performance of Local Memory Copy 58
2.5.2 Performance of Direct Transfers to Remote Memory 58
2.5.3 Performance of MPI/PVM Transfers 61
2.5.4 Performance of TCP/IP Transfers 64
2.5.5 Discussion and Comparison 65
2.6 Summary 67

Part II. SCI Hardware

3. Dolphin SCI Adapter Cards
Marius Christian Liaaen, Hugo Kohmann 71

3.1 Introduction 71
3.2 Overview of the Adapter Cards 71
3.3 Operating Modes of the SCI Cards 73
3.4 SCI Requester 74
3.4.1 Address Mapping 74
3.4.2 SCI Transaction Handling 75
3.4.3 SCI Packet Requester 77
3.5 SCI Responder 78
3.5.1 Mailbox 79
3.5.2 Access Protection 79
3.5.3 Atomic Access 79
3.5.4 Host Bridge Capabilities 80
3.6 DMA Transfers 80
3.6.1 DMA Transfers on the SBus Card 80
3.6.2 DMA Transfers on the PCI Card 80
3.7 Interrupter 81
3.8 Concurrency Issues 81
3.8.1 Write Assembly 81
3.8.2 Efficient Store Barrier 81
3.9 Performance 82
3.10 Applications and Topologies 82
3.10.1 SAN Interface Adapter 83
3.10.2 Remote I/O Connection and Data Acquisition 83

3.10.3 Switches and Topologies 83
3.11 Cluster Software 85

4. The TUM PCI/SCI Adapter
Georg Acher, Wolfgang Karl, Markus Leberecht 89

4.1 Introduction 89
4.2 The PCI/SCI Adapter Architecture 90
4.3 SCI Packet Encoding and Decoding 92
4.3.1 Overview of Packet Processing 92
4.3.2 Choosing the Technology 92
4.3.3 Internal Structure of the FPGA 93
4.3.4 Structure of the Packet Manager as a Microcode Sequencer 95
4.3.5 Microcode Examples 97
4.3.6 Benefits of the Micro Sequencer 98
4.4 The SCI Unit 99
4.5 Preliminary Results for the PCI/SCI Adapter 99
4.6 Related Work 100
4.7 Conclusion 100

Part III. Interconnection Networks with SCI

5. Low-Level SCI Protocols and Their Application to Flexible Switches
Andreas C. Döring, Wolfgang Obelöer, Gunther Lustig, Erik Maehle 105

5.1 Introduction 105
5.2 Data Format of SCI Packets 105
5.3 Flow Control 107
5.3.1 Flow Control in Rings 107
5.3.2 Packet Sequence in SCI 108
5.3.3 Determination of State Transitions 109
5.4 Bandwidth Multiplexing 110
5.4.1 Bandwidth Management in One Ring 110
5.4.2 Idle Symbols 112
5.4.3 Time-Out Determination 113
5.5 Network Interface 113
5.5.1 Requirements 114
5.5.2 Products 114
5.6 Routers 115
5.6.1 Requirements 115
5.6.2 Products and Challenges 116
5.6.3 Flexible Router 117
5.6.4 Strip-off Decision 118

5.6.5 Routing Decision and Topology ... 119
5.7 Rule-Based Routing ... 120
5.8 Conclusion and Outlook ... 121

6. SCI Rings, Switches, and Networks for Data Acquisition Systems
Harald Richter, Richard Kleber, Matthias Ohlenroth ... 125

6.1 Introduction ... 125
6.2 SCI-based Data Acquisition Systems ... 126
6.3 SCINET Test Beds ... 127
6.4 Measurement Results ... 129
6.5 SCI Switches ... 134
6.6 Efficient Use of SCI Switches ... 136
6.7 Multistage SCI Networks ... 139
6.8 Simulation Results ... 141
6.9 Summary and Conclusions ... 146

7. Scalability of SCI Ringlets
Geir Horn ... 151

7.1 Do SCI Ringlets Scale in Number of Nodes? ... 151
7.2 Ringlet Bandwidth Model ... 152
7.2.1 Transaction Formats ... 152
7.2.2 Packet Generation ... 155
7.2.3 Address Distribution ... 155
7.2.4 Locality ... 156
7.2.5 Bypass Rate ... 157
7.2.6 Echo Packet Rate ... 158
7.2.7 Output Link Utilization Factor ... 160
7.3 Scalability Evaluation ... 160
7.3.1 Common Assumptions ... 161
7.3.2 Uniform Ringlet Traffic ... 162
7.3.3 Non-uniform Ringlet Traffic ... 162
7.3.4 Changing Packet Lengths ... 163
7.4 Discussion ... 163
7.5 Conclusion ... 165

8. Affordable Scalability Using Multi-Cubes
Håkon Bugge, Knut Omang ... 167

8.1 Introduction ... 167
8.2 Interconnect Overview ... 168
8.3 Methodology ... 168
8.4 Analysis ... 170
8.4.1 “Hot-Link” Analysis ... 170

8.4.2 "Hot-B-Link" Analysis ... 171
8.5 Results ... 172
8.6 Conclusions ... 174

Part IV. Device Driver Software and Low-Level APIs

9. Interfacing SCI Device Drivers to Linux
Roger Butenuth, Hans-Ulrich Heiss ... 179

9.1 Introduction ... 179
9.2 Layers of Functionality ... 180
9.2.1 Address Spaces ... 180
9.2.2 Levels of Hardware Abstraction ... 180
9.2.3 Resource Management ... 182
9.2.4 Virtual Mapping ... 183
9.2.5 Robustness ... 184
9.3 Why Linux? ... 185
9.4 Interfaces of the Driver ... 186
9.4.1 Hardware ... 186
9.4.2 Linux ... 187
9.4.3 User Processes ... 188
9.4.4 SCI Drivers on Other Nodes ... 188
9.5 Conclusions ... 189

10. SCI Physical Layer API
Volker Lindenstruth, David B. Gustavson ... 191

10.1 Introduction ... 191
10.1.1 Scope of the Standard ... 192
10.2 SCI Physical Layer API Architecture and Features ... 193
10.2.1 Exception Handling ... 195
10.2.2 Endianness ... 195
10.3 Supported Data Types ... 196
10.4 Miscellaneous Procedures ... 196
10.5 Address Translation Model ... 197
10.5.1 Global Object Identifier ... 199
10.5.2 SCI Global Address Resolution ... 200
10.6 Shared Memory Transactions ... 200
10.7 Packet Transactions ... 202
10.8 Block Transactions ... 202
10.9 Message Passing Transactions ... 203
10.10 Cache Transactions ... 204
10.11 Conclusions ... 205

Part V. Message Passing Libraries

11. SCI Sockets Library
Hermann Hellwagner, Josef Weidendorfer 209

11.1 Introduction 209
11.1.1 Rationale 209
11.1.2 Overview 210
11.2 Features and Design 210
11.2.1 Features 210
11.2.2 Components 211
11.2.3 Communication via the SSLib 212
11.2.4 Connection Setup 214
11.2.5 Handling Special System Calls 216
11.2.6 Other Calls Intercepted and Handled by the SSLib 218
11.2.7 Out-of-Band Data 218
11.3 Implementation Aspects 218
11.3.1 Communication Among Components 218
11.3.2 SSLib Layers 219
11.3.3 Choice of Most Efficient Communication Mechanism 220
11.3.4 SSLib Implementations 221
11.3.5 Control Transfers 221
11.4 Functional Tests and Performance 222
11.5 Related Work 224
11.6 Conclusions 227

12. TCP/IP over SCI under Linux
Hüseyin Taskin, Roger Butenuth 231

12.1 Introduction 231
12.2 SCIP Structure 232
12.2.1 Packet Driver Interface 232
12.2.2 Hardware Address Resolution 232
12.2.3 Other Implementation Issues 233
12.3 Performance 234
12.3.1 Configuration 234
12.3.2 Latency 234
12.3.3 Throughput 235
12.4 Conclusion 237

13. PVM for SCI Clusters
Markus Fischer, Alexander Reinefeld 239

13.1 Overview 239
13.2 Parallel Virtual Machine 239

13.2.1 PVM Implementations 240
13.2.2 Models for Zero-Memory-Copy Data Transfer 241
13.3 SCI Communication Model 242
13.4 PVM-SCI 243
13.4.1 System Architecture 243
13.4.2 Supporting Multiple Interconnects 245
13.4.3 Reducing Memory Copies 245
13.4.4 Ring Buffer Management 246
13.4.5 Performance Results 247
13.5 Conclusions 247

14. ScaMPI – Design and Implementation
L.P. Huse, K. Omang, H. Bugge, H. Ry, A.T. Haugsdal, E. Rustad 249

14.1 Introduction 249
14.2 Scali Systems 249
14.3 The SCI Memory Model 250
14.3.1 Coordinating Use of Shared Locations 251
14.3.2 Ensuring Safe Data Transport in SCI – Checkpointing 252
14.3.3 Shared Address Space Programming without the Drawbacks 252
14.4 ScaMPI Design Goals 253
14.5 ScaMPI Implementation 254
14.5.1 Fault Tolerance 254
14.5.2 User Friendliness 256
14.5.3 Third Party Software 256
14.6 Performance Results 257
14.6.1 Barrier 258
14.6.2 All-to-All Communication 259
14.7 Conclusions 260

Part VI. Shared Memory Programming Models and Runtime Mechanisms

15. Shared Memory vs Message Passing on SCI: A Case Study Using Split-C
Max Ibel, Michael Schmitt, Klaus Schauser, Anurag Acharya 267

15.1 Introduction 267
15.1.1 Introduction to Split-C 268
15.1.2 Introduction to Active Messages 269
15.2 Message-Passing Implementation 269
15.2.1 Active Messages on Top of SCI 269
15.2.2 Split-C on Top of Active Messages 272
15.3 Shared Memory Implementation 273

15.3.1 Split-C on Top of SCI 273
15.4 Experimental Evaluation 274
15.4.1 Micro-benchmarks 274
15.4.2 Application Benchmarks 276
15.5 Hybrid Implementation 277
15.5.1 Basic Framework 277
15.5.2 Mapping Strategies 278
15.6 Conclusions 279

16. A Shared Memory Programming Interface for SCI Clusters
Marcus Dormanns, Karsten Scholtyssik, Thomas Bemmerl 281

16.1 Introduction 281
16.2 Platform Properties: System Image and Memory Model 282
16.2.1 System Image and Operational Model 282
16.2.2 Memory Model 283
16.3 User Front-End 284
16.4 The Application Programmer's Interface 284
16.4.1 Initialization and Execution Environment 286
16.4.2 Memory Management 286
16.4.3 Synchronization 288
16.4.4 Loop Scheduling 288
16.5 Conclusions 289

17. True Shared Memory Programming on SCI-based Clusters
Martin Schulz 291

17.1 Introduction 291
17.2 Designing a Global Virtual Memory 292
17.2.1 Building Block 1: SCI-based Hardware DSM 292
17.2.2 Building Block 2: Software DSM Systems 293
17.2.3 Combining Both Building Blocks to the SCI-VM 293
17.2.4 Locality Issues and Caching 294
17.3 SCI-VM Implementation Challenges 295
17.3.1 Mapping of Individual Page Frames 295
17.3.2 Dynamically Paged Memory 296
17.3.3 Enabling Caching Using Relaxed Consistency 296
17.4 Framework for SCI-VM-based Programming Models 297
17.4.1 SCI-VM Interface 297
17.4.2 Tradeoff Between Transparency and Performance 298
17.4.3 Current Status of the Framework 298
17.5 SPMD Programming Model on Top of SCI-VM 299
17.5.1 The Execution Model 299
17.5.2 Allocating Shared Memory 300

17.5.3 Synchronization 300
17.5.4 Consistency Model 301
17.6 Experiments and Results 302
17.6.1 Experimental Setup 302
17.6.2 Results for the Numerical Kernels 302
17.6.3 Results for the Volume Rendering Code 304
17.7 Using the SCI-VM for Transparent Multithreading 305
17.7.1 Transparent Thread Distribution 305
17.7.2 Synchronization Mechanisms 306
17.7.3 Applying a Relaxed Consistency Model 306
17.8 Related Work 307
17.9 Conclusions and Future Work 308

18. Implementing a File System Interface to SCI
P.T. Koch, J.S. Hansen, E. Cecchet, X. Rousset de Pina 313

18.1 Introduction 313
18.1.1 Motivation 313
18.1.2 SCI-based File Systems 314
18.1.3 Outline 314
18.2 Sharing in File Systems 315
18.2.1 Memory-Mapped Files 315
18.2.2 UNIX Example with a Memory-Mapped File 316
18.2.3 File Consistency 316
18.2.4 Synchronization 317
18.3 Issues for Implementing SCI-based File Systems 317
18.3.1 A Virtual File System 318
18.3.2 Files and Directories 319
18.3.3 Example of Vnode/vfs Data Structures 319
18.3.4 Virtual File System Operations 320
18.3.5 Interaction with the Virtual Memory System 321
18.3.6 Remote Memory Mappings and File Consistency 322
18.3.7 Synchronization 322
18.4 The SciOS Prototype 323
18.4.1 SciOS Memory Protocols 323
18.4.2 Main File System Data Structures 324
18.4.3 The GLOBAL Memory Protocol 325
18.4.4 Memory Protocol Implementation in Linux 327
18.5 Related Work 328
18.6 Summary and Conclusions 329

19. Programming SCI Clusters Using Parallel CORBA Objects
Thierry Priol, Christophe René, Guillaume Alléon 333

19.1 Introduction 333

19.2 Parallel vs. Distributed Programming 333
19.3 An Overview of CORBA 335
19.4 Parallel CORBA Objects 336
19.4.1 Execution Model 336
19.4.2 Extended-IDL 337
19.4.3 Implementation of Parallel CORBA Objects 340
19.5 The *Cobra* Runtime System 340
19.5.1 *Cobra* Services 341
19.5.2 *Cobra* Software Architecture 342
19.6 A Case Study: The IDAHO Application 344
19.7 Related Work 346
19.8 Conclusion and Perspectives 347

20. The MuSE Runtime System for SCI Clusters: A Flexible Combination of On-Stack Execution and Work Stealing
Markus Leberecht 349

20.1 Introduction 349
20.2 The MuSE System 351
20.2.1 The SMiLE Cluster of PCs 351
20.2.2 The Multithreaded Scheduling Environment 352
20.3 Experimental Evaluation 357
20.3.1 Basic Runtime System Performance 357
20.3.2 Load Balancing and Parallelism Generation 358
20.4 Related Work and Conclusion 362

Part VII. Benchmark Results and Application Experiences

21. Large-Scale SCI Clusters in Practice: Architecture and Performance
Jens Simon, Alexander Reinefeld, Oliver Heinz 367

21.1 Introduction 367
21.2 PSC System Architecture 367
21.2.1 Node Configuration 368
21.2.2 SCI Interconnect 369
21.2.3 Software Configuration 370
21.3 Standard Benchmarks 371
21.3.1 Low-Level MPI Benchmarks 371
21.3.2 Parallel Linpack 373
21.3.3 FFT Benchmarks 374
21.3.4 HINT Benchmark 375
21.4 Applications 376
21.5 Summary 379

22. Shared Memory Parallelization of the GROMOS96 Molecular Dynamics Code
Marcus Dormanns 383

22.1 Introduction 383
22.2 The GROMOS Code 384
22.2.1 General Code Characteristics 384
22.2.2 Structure of the Code 384
22.3 Parallelization 386
22.3.1 Starting with Parallelism and Coordinating I/O 386
22.3.2 Parallelization of the Interaction Calculation Kernels . . 387
22.4 Performance Results 392
22.4.1 Hardware Platform 392
22.4.2 Results 392
22.4.3 Performance Comparison to Other Parallel GROMOS Implementations 393
22.5 Conclusion 394

23. SCI Prototyping for the Second Level Trigger System of the ATLAS Experiment
A. Belias, A. Bogaerts, D. Botterill, J. Dawson, E. Denes, F. Giacomini, R. Hauser, C. Hortnagl, R. Hughes-Jones, S. Kolya, D. Mercer, R. Middleton, J. Schlereth, P. Werner, F. Wickens 397

23.1 Introduction 397
23.2 The ATLAS Trigger System 397
23.3 Low-Level API 399
23.3.1 Basic Performance Measurements 400
23.4 The ATLAS Level-2 Trigger Demonstrator 403
23.4.1 Hardware 405
23.4.2 Software 406
23.4.3 Vertical Slice Configurations 407
23.4.4 Conclusions 410
23.5 Objectives and Design of the Second Prototype 410
23.5.1 Lessons Learned from the Demonstrator 410
23.5.2 Testbed 411
23.5.3 Software 413
23.5.4 SCI Testbed 413

Part VIII. Tools for SCI Clusters

24. SCI Monitoring Hardware and Software: Supporting Performance Evaluation and Debugging
Wolfgang Karl, Markus Leberecht, Michael Oberhuber 417

24.1 Introduction 417
24.2 The Monitoring Approach for the SMiLE PC Cluster 418
24.3 The Controlled Deterministic Execution Approach (*CODEX*) 422
24.4 Controlling Execution with SMiLE 425
24.4.1 Mapping POEM to the SMiLE Architecture 425
24.4.2 Controlling Execution on SMiLE 426
24.4.3 A Framework for an Implementation of *CODEX* for Fine-Grained DSM Execution 428
24.5 Related Work 430
24.6 Conclusion 430

25. Monitoring SCI Clusters
Matthias Maier-Stahel, Roger Butenuth, Hans-Ulrich Heiss 433

25.1 Motivation 433
25.2 General Architecture 434
25.3 Monitor Agents 434
25.4 Master 436
25.5 Visualizer 437
25.6 Conclusion 440

26. Multi-User System Management on SCI Clusters
Matthias Brune, Axel Keller, Alexander Reinefeld 443

26.1 Introduction 443
26.1.1 Hardware Scenario 443
26.1.2 Software Scenario 444
26.1.3 User Access and System Management 445
26.2 Architecture of CCS 445
26.2.1 Island Concept 445
26.2.2 User Interface 446
26.2.3 Scheduling 448
26.2.4 Partitioning the System 450
26.2.5 Job Creation and Control 451
26.2.6 Reliability 453
26.3 Resource and Service Description 454
26.3.1 Graphical Representation 455
26.3.2 Textual Representation 456
26.3.3 Internal Data Representation 457

26.4 Related Work 459
26.5 Summary 459

Part IX. Perspectives

27. Industrial Takeup of SCI and Future Developments
David B. Gustavson 465

27.1 SCI's Cultural Context 465
27.2 SCI Marketing and Adoption 469
27.3 Commercial Adoption of SCI 472
27.3.1 Interface Chips and Products 472
27.3.2 Coherent Shared Memory Implementations 473
27.3.3 Non-coherent Implementations 475
27.4 Future Directions 476
27.4.1 IEEE P2100 (SerialPlus) 477
27.4.2 Concurrent Buses—A New Name for this Technology . 478
27.4.3 Concurrent Behavior is Essential for Scalability 479

List of Contributors 481

Subject Index 487

Part I

SCI and Competitive Interconnects for Cluster Computing

Scalable Coherent Interface (SCI) is an innovative, comprehensive, high-performance interconnect technology providing solutions for a wide range of applications. The focus of this book is on one of these applications, namely compute clusters based on multi-Gigabit SCI cluster networks.

To put this application and the articles in this book into perspective, this first part of the book provides an introduction to the rationale, important concepts and protocols, benefits and limitations, and the wealth of potential applications of the SCI interconnect standard. SCI is extremely flexible, with its applications ranging from compute and server clusters, to highly scalable, cache-coherent shared-memory multiprocessors, distributed I/O systems, and novel ways of coupling processors and memories.

SCI also comprises a whole family of standards; even the base standard can be viewed as covering three standards: the physical layer, the logical layer, and the cache-coherence protocols of a shared-memory interconnect. Unfortunately—probably due to the complexity of the specification—SCI has not achieved its goal of becoming a truly open, widely accepted interconnect system that allows diverse devices by multiple vendors to be readily plugged in, as enabled by open bus specifications, for instance. Rather, proprietary implementations in many of the application areas have emerged, with vendors making changes and extensions to the base SCI specification as suited to their needs.

Chapter 1 briefly covers these aspects of SCI. It provides an introduction to the core SCI technology underlying this book, alludes to compatible extensions of the SCI standard, and describes "classical" applications of SCI. The latter comprise the use of SCI as a system area network for clusters, as a memory interconnect for scalable cache-coherent multiprocessors, as an I/O subsystem interconnect, and as the communication infrastructure for high-performance data acquisition systems. Examples of existing products and research prototypes are given. Furthermore, related Gigabit interconnection networks and concepts are briefly reviewed.

H. Hellwagner, . Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 1-2, 1999
© Springer-Verlag Berlin Heidelberg 1999

Chapter 2 goes into the technical details and provides interesting performance results of a comparison of several Gigabit network technologies, predominantly the Dolphin SCI cluster interconnect and the Myricom Myrinet system area network. The SGI/Cray T3D MPP network and its performance are given as a reference point, as well as results on ATM and Gigabit Ethernet representing the mainstream network technologies. A comprehensive benchmark suite has been worked out for this investigation, covering multiple communication software layers (raw data transfers, MPI-style and TCP protocol-based communication) and aspects (e.g., transfer of data stored non-contiguously in memory). The results disclose benefits and limitations of the SCI-based Dolphin cluster interconnect and provide a thorough quantitative comparison of important Gigabit networks.

1. The SCI Standard and Applications of SCI

Hermann Hellwagner

Institute of Information Technology, University of Klagenfurt
A–9020 Klagenfurt, Austria
email: hermann.hellwagner@uni-klu.ac.at
http://www.itec.uni-klu.ac.at/

1.1 Introduction

Scalable Coherent Interface (SCI) is an innovative interconnect standard (ANSI/IEEE Std 1596-1992 [26]) that addresses the high-performance computing and networking domain. This book describes in depth one specific application of SCI, namely its use as a high-speed interconnection network (often called a system area network) for compute clusters built from commodity workstation nodes. Yet, SCI's original design does not specifically address this segment and several other applications of SCI have been devised and realized.

This chapter therefore provides the context for the rest of the book by (1) introducing the basic concepts of SCI and related standards, and (2) presenting the most important other uses and implementations of SCI not covered by the contributions in this book.

In doing so, we will not delve deeply into the technology of SCI and its various implementations, but rather concentrate on the essential concepts so that the reader may acquire the basic knowledge required for the subsequent chapters and perceive the flexibility and wide applicability of SCI.

For technical details, the interested reader is referred to the literature pertaining to this chapter, most importantly the SCI standard document [26] and several survey articles [16][18]. When needed, later chapters will provide more background information on SCI, e.g., Chapter 5 on low-level protocols.

This introductory chapter is organized as follows. Section 1.2 summarizes the principal objectives and features of SCI and attempts to assess the achievements and current status of SCI. Section 1.3 presents essential concepts of the base SCI standard in some more detail and refers to some of the extensions developed so far or currently under discussion. Section 1.4 outlines four application areas of SCI, namely its use as:

– a system area network for compute clusters, which will be treated in depth in the remainder of this book;
– a memory interconnect for large-scale cache-coherent multiprocessors;
– an I/O subsystem interconnect; and
– an interconnect for high-performance data acquisition systems.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 3-37, 1999
© Springer-Verlag Berlin Heidelberg 1999

Examples of existing systems are given there. Related interconnection networks and concepts are dealt with in Section 1.5. Concluding remarks are given in Section 1.6. Other uses and future developments of SCI are also addressed in Chapter 27.

1.2 SCI Overview

1.2.1 Background

SCI has its origins in an effort of bus experts in the late 1980s to define a very high performance computer bus ("Superbus") that would support a significant degree of multiprocessing (i.e., number of processors). However, the group soon realized that backplane bus technology would not be able to meet this requirement, despite advanced concepts like split transactions and sophisticated, expensive implementations of the latest bus standards and products.

The reasons are that (1) a bus is a centralized resource and, thus, an inherent serial bottleneck that would be exacerbated by ever faster microprocessors, and (2) bus signaling is already approaching its fundamental limits (speed of light), resulting in electrically complex and expensive solutions and in short bus lengths. Both factors limit scalability to a few state-of-the-art, high-speed microprocessors that can be well served by a common bus.

As a result, the committee abandoned the bus-oriented view and developed novel, distributed solutions to overcome the shared-resource and signaling problems, while retaining the overall goal of defining an interconnect that offers the convenient services well known from centralized buses.

The resulting specification, SCI, finally approved in 1992, thus describes hardware and protocols that provide processors with the shared-memory view of buses. SCI also specifies related transactions to read, write, and lock memory locations without software protocol involvement, as well as to transmit messages and interrupts. Hardware protocols to keep processor caches coherent are defined as an implementation option. In contrast to most previous solutions, the SCI interconnect, the memory system, and the associated protocols are fully distributed and scalable: an SCI network is based on point-to-point links only, implements a distributed shared memory (DSM) in hardware, and avoids serialization in almost any respect.

1.2.2 Goals

Several ambitious goals guided the specification process of SCI and partly determined its name.

High performance. The primary objective of SCI, as of any interconnection network and associated protocols, is to deliver high communication performance to parallel or distributed applications. This comprises three aspects [14]:

- high sustained *throughput*;
- low *latency*;
- low CPU *overhead* for communication operations.

The performance goals set forth were in the range of GBit/s link speeds and latencies in the low microseconds range in loosely-coupled systems, even less in tightly-coupled multiprocessors.

Scalability. SCI was devised to address scalability in many respects, among them [17]:

- scalability of *performance* (aggregate bandwidth) as the number of nodes attached to the system grows;
- scalability of interconnect *distance*, from centimeters to tens or even hundreds of meters, depending on the media and physical layer implementation, yet based on the same logical layer protocols;
- scalability of the *memory system*, in particular of the *cache coherence protocols*, which must not have a built-in limit on the number of processors or memory modules that could be handled;
- *technological* scalability, i.e., (1) use of the same mechanisms in large-scale and small-scale as well as tightly-coupled and loosely-coupled systems, and (2) the ability to readily make use of advances in technology, e.g., high-speed links;
- *economic* scalability, i.e., use of the same mechanisms and components in low-end, high-volume and high-end, low-volume systems, opening the possibility to leverage the economies of scale of mass production of SCI hardware;
- no short-term practical limits to the *addressing capability*, i.e., an addressing scheme for the DSM wide enough to support a large number of nodes and a large memory on each node.

Coherent memory system. Caches are becoming ever more important for modern microprocessors to reduce average access time to data. This specifically holds for a DSM system with NUMA characteristics (non-uniform memory access) where remote accesses can be roughly an order of magnitude more expensive than local ones. To support a convenient programming model as known, e.g., from symmetric multiprocessors (SMPs), the caches should be kept coherent in hardware.

Interface characteristics. The SCI specification was intended to describe a standard *interface* to an interconnect that would enable multiple devices from multiple vendors to be attached and to interoperate. In other words, SCI should serve as an *open distributed bus* connecting components like microprocessors, memory modules, and intelligent I/O devices in a high-speed system.

1.2.3 Concepts

Many of the goals have been met, but some have not. In the following, the major concepts and features of SCI will be summarized and an attempt will be made to assess the achievements and the current status of SCI, as represented by several implementations of SCI networks.

Point-to-point links. An SCI interconnect is defined to be built only from unidirectional, point-to-point links between participating nodes. These links can be used for concurrent data transfers, in contrast to the one-at-a-time communication characteristics of buses. The number of the links grows as nodes are added to the system, increasing the aggregate bandwidth of the network. The links can be made fast and their performance can scale with improvements in the underlying technology. They can be implemented in a bit parallel manner (for small distances) or in a bit serial fashion (for larger distances), with the same logical-layer protocols.

Most implementations today use parallel links over distances of a few centimeters or meters.

Sophisticated signaling technology. The data transfer rates and lengths of shared buses are inherently limited due to signal propagation delays and signaling problems on the transmission lines, such as capacitive loads that have to be driven by the sender, impedance mismatches, and noise and signal reflections on the lines. The unidirectional, point-to-point SCI links avoid these signaling problems. Since there is only a single transmitter and a single receiver rather than multiple devices (capacitive loads), the signaling speed can be increased significantly. High speeds are also fostered by low-voltage differential signals; see Section 1.3.3.

Furthermore, SCI strictly avoids back-propagating signals, even reverse flow control on the links, in favor of high signaling speeds and scalability. (A reverse flow control signal would make timing of, and buffer space required for, a link dependent on the link's distance [18].) Thus, flow control information becomes part of the normal data stream in the reverse direction, leading to the requirement that an SCI node (as described below) must at least have one outgoing link and one incoming link. Many of these link-level issues and low-level protocols are discussed in Chapter 5.

SCI link speeds today reach 500 MByte/s in system area networks (distances of a few meters, 16-bit parallel links, clocked at 125 MHz, differential signals, CMOS technology) [9] and 1 GByte/s in closely-coupled, cache-coherent shared-memory multiprocessors (GaAs link controller) [41]; transfer rates of 1 GByte/s have also been demonstrated over a distance of about 100 meters, using parallel fiber-ribbon cables and BiCMOS link-level devices [13].

Nodes. SCI was designed to connect a large number of *nodes* (up to 64k). A node can be a complete workstation or server machine, a processor and its associated cache only, a memory module, I/O controllers and devices, or bridges to other buses or interconnects, as illustrated exemplarily in Figure 1.1.

Each node is required to have a standard interface to attach to the SCI network, as described in Section 1.3. In most SCI systems implemented so far, nodes are complete machines, often even multiprocessors.

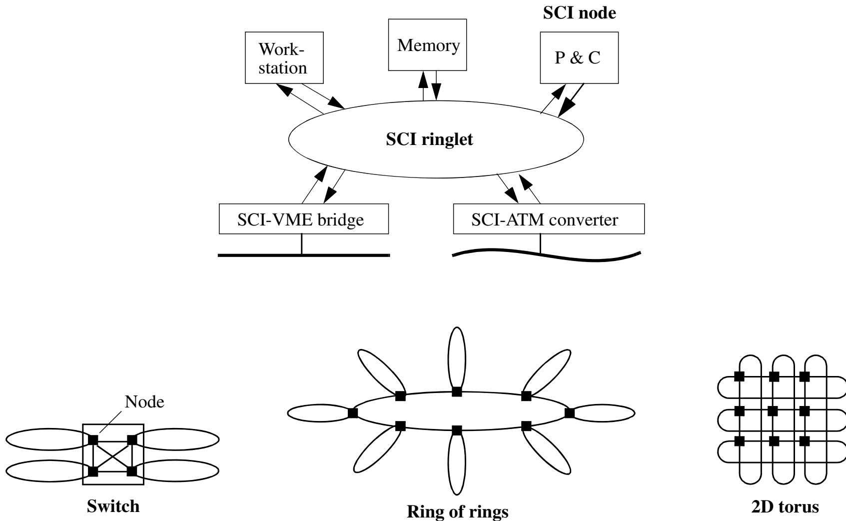


Fig. 1.1. Simple SCI network topologies

Topology independence. In principle, SCI networks with complex topologies could be built; investigations into this area are described, e.g., in Chapters 6 and 8. However, the standard anticipates simple topologies to be used. For small systems, for instance, the preferred topology is a small ring (a so-called *ringlet*); for larger systems, topologies like a single switch connecting multiple ringlets, rings of rings, or multidimensional tori are feasible; see Figure 1.1.

Most SCI systems to date use single rings, a switch, multiple rings, or two-dimensional tori. Special applications with well-known communication patterns or very high bandwidth requirements may require specific multistage topologies to be devised, as shown exemplarily for a data acquisition system in Chapter 23.

Fixed addressing scheme. SCI uses the 64-bit fixed addressing scheme defined by the Control and Status Register (CSR) Architecture standard (IEEE Std 1212-1991) [25]. The 64-bit SCI address is divided into two fixed parts: the most significant 16 address bits specify the node ID (node address) so that an SCI network can comprise up to 64k nodes; the remaining 48 bits are used for addressing *within* the nodes, in compliance with the CSR Architecture standard.

While this addressing model meets the requirement of ample addressing capability, conversion from the 32-bit address space that most nodes employ today, to the 64-bit global SCI address space becomes necessary. An example of this address transformation is given in Chapter 3.

Hardware-based distributed shared memory (DSM). The SCI addressing scheme spans a global, 64-bit address space; in other words, a physically addressed, distributed shared memory system. The distribution of the memory is transparent to software and even to processors, i.e., the memory is logically shared, just as in a system with a centralized bus and shared memory. A memory access by a processor is mediated to the target memory module by the SCI hardware.

The major advantage of this feature is that inter-node communication can be effected by simple load and store operations by the processor, without invocation of a software protocol stack. The instructions accessing remote memory can be issued at *user level*; the operating system need not be involved in communication. This results in very low latencies for SCI communication, typically in the low microseconds range.

A major implementation challenge, however, is how to integrate the SCI network (and, thus, access to the system-wide SCI DSM) with the memory architecture of a standard single-processor workstation or a multiprocessor node. Sections 1.4.1 and 1.4.2 will describe the common solutions – attaching SCI to the I/O bus or to the memory bus – in more detail.

Bus-like services. To complete the hardware DSM, SCI defines transactions to read, write, and lock memory locations, functionality well-known from computer buses. In addition, message passing and global time synchronization are supported, both as defined by the CSR Architecture; interrupts can be delivered remotely as well. Broadcast functionality is also defined.

Transactions can be tagged with four different priorities. In order to avoid starvation of low-priority nodes, fair protocols for bandwidth allocation and queue allocation have been developed. Bandwidth allocation is similar in effect to bus arbitration in that it assigns transfer bandwidth (if scarce) to nodes willing to send. Queue allocation apportions space in the input queues of heavily loaded, shared nodes, e.g., memory modules or switch ports which are targeted by many nodes simultaneously. Since the services have to be implemented in a fully distributed fashion, the underlying protocols are rather complex.

Split transactions. Like modern multiprocessor buses, SCI strictly splits its transactions into *request* and *response* phases. This is a vital feature to avoid scalability impediments; that is, it makes signaling speed independent of the distance a transaction has to travel and avoids monopolizing network links. Transactions therefore have to be self-contained and are sent as packets, containing a transaction ID, addresses, commands, status, and data as needed. A consequence is that multiple transactions can be outstanding

per node. Transactions can thus be pumped into the network at a high rate, using the interconnect in a pipeline fashion.

Most SCI systems today do make use of this feature in order to achieve high throughput rates, yet not to the full extent of 64 outstanding transactions as defined in the standard. The decoupled nature of transactions again adds complexity due to the need of keeping track of outstanding packets and more complicated flow control.

Optional cache coherence. SCI defines distributed cache coherence protocols, based on a distributed-directory approach (doubly-linked sharing list per shared, cacheable memory block), a multiple readers–single writer sharing regime, and write invalidation [24]. The memory coherence model is purposely left open to the implementor, allowing sequential consistency or more relaxed memory models to be realized in SCI systems. The standard provides optimizations for common situations such as pairwise sharing that improve performance of frequent coherence operations.

The cache coherence protocols are designed to be implemented in hardware; however, they are highly sophisticated and complex. The complexity stems from a large number of states of coherent memory and cache blocks, correspondingly complex state transitions, and the advanced algorithms that ensure atomic modifications of the distributed sharing lists (e.g., insertions, deletions, invalidations). The greatest complication arises from the integration of the SCI coherence protocols with the snooping protocols typically employed on the nodes' memory buses. Although the standard specifies three sets of coherence implementation options (the minimal set, a typical set, and the full set), the implementation is highly challenging and incurs some risks and potentially high costs. Not surprisingly, only a few companies have done implementations so far; see Section 1.4.2.

The cache coherence protocols are provided as *options* only. A compliant SCI implementation need not cover coherence; an SCI network even *cannot* participate in coherence actions when it is attached to the I/O bus as is the case in the compute clusters addressed in this book; see Section 1.4.1. Yet, a common misconception has emerged over the years that cache coherence is required functionality at the core of SCI. This misunderstanding has clearly hindered SCI's fast proliferation for non-coherent uses, e.g., as a system area network or a high-speed LAN.

Reliability in hardware. In order to enable high-speed transmission, error detection is done in hardware, based on a 16-bit CRC code which protects each SCI packet. Transactions and hardware protocols are provided that allow a sender to detect failure due to packet corruption, and allow a receiver to notify the sender of its inability to accept packets (due to a full input queue) or to ask the sender to re-send the packet. Since this happens on a per-packet basis, SCI does not automatically guarantee in-order delivery of packets. This may have a considerable impact on software which would rely on a guaranteed packet sequence. An example is a message passing library

delivering data into a remote buffer using a series of remote write transactions and finally updating the tail pointer of the buffer; see Chapter 11, for instance. SCI hardware like Dolphin's SCI cluster adapter provides functionality to enforce a certain memory access order, e.g., via memory barrier operations; see Chapter 3.

Various time-outs are provided to detect lost packets or transmission errors. Hardware retry mechanisms or software recovery protocols may be implemented based on the standard transmission-error detection and isolation mechanisms; these are however not part of the standard. As a consequence, SCI implementations today differ widely in the way errors are dealt with.

The protocols are designed to be robust, i.e., they should, for instance, survive the failure of a node with outstanding transactions. For this purpose, error containment and logging procedures, ringlet maintenance functions and a packet time-out scheme are specified, among other mechanisms. Robustness is particularly important for the cache coherence protocols which are designed to behave correctly even if a node fails amidst the modification of a distributed sharing list.

Layered specification. The SCI specification is structured into three layers: the physical layer, the logical layer, and the optional cache coherence layer. The latter two layers will be dealt with in more detail in Section 1.3.

At the physical layer, three initial physical link models are defined: a parallel electrical link operating at 1 GByte/s over short distances (meters); a serial electrical link that operates at 1 GBit/s over intermediate distances (tens of meters); and a serial optical link that operates at 1 GBit/s over long distances (kilometers).

Although the definitions include the electrical, mechanical, and thermal characteristics of SCI modules, connectors, and cables, the specifications were not adhered to in the early implementations. SCI systems today typically use specific physical layer implementations, incompatible to others. It is fair to say, therefore, that SCI has not become the open, distributed interconnnect system that the designers had envisaged to create.

C code. One of the most remarkable features of the SCI standard is that major portions are provided in terms of a "formal" specification, namely C code. (Exceptions are packet formats and the physical layer specifications.) Text and figures are considered explanatory only, the definitive specification are the C listings. The major reasons for this approach are that C code is (largely) unambiguous and not easily misunderstood and that the specification becomes executable, as a simulation. In fact, much of the specification was validated by extensive simulations before release. This was deemed necessary because SCI introduces new approaches and many novel, sophisticated distributed protocols. The designers could not always be sure that their solutions were correct and feasible.

1.2.4 Discussion

SCI addresses difficult interconnect problems and specifies innovative distributed structures and protocols for a scalable distributed shared memory architecture. In doing so, the designers have come to cover a wide spectrum of bus, network, and memory architecture problems, ranging from signaling considerations up to distributed directory-based cache coherence mechanisms.

In fact, this wide scope of SCI has raised criticism that the standard is actually "several standards in one" and difficult to understand and work with. This lack of a clear profile and the wide applicability of SCI have probably contributed to its relatively modest acceptance in industry.

Furthermore, as pointed out above, the SCI protocols are complex (albeit well-devised) and not easily implemented in silicon. Thus, implementations are quite complex and therefore expensive. (Yet, at the physical and logical levels, they are simple compared to any split-transaction bus.)

In an attempt to reduce complexity and optimize speed, many of the implementors adopted the concepts and protocols which they regarded as appropriate for their application, and left out or changed other features according to their needs. This use of SCI led to a number of proprietary, incompatible implementations.

As a consequence, the goal of economic scalability has not been satisfactorily achieved in general. (However, this does not necessarily hold for individual companies which may well leverage the economies of scale for their own implementation to achieve reasonable prices for their SCI products.)

As a further consequence, SCI has clearly also missed the goal of evolving into an open distributed "bus" that multiple devices from different vendors could attach to and interoperate.

However, in terms of the technical objectives, predominantly high performance and scalability, SCI has well achieved its ambitious goals. The vendors that have adopted and implemented SCI (in various flavors), offer innovative high-throughput, low-latency interconnect products (see Sections 1.4.1 and 1.4.3) or full-scale shared-memory multiprocessing systems (see Section 1.4.2).

1.3 The SCI Standard and Some Extensions

This section describes the most important concepts of SCI in further detail, with a focus on the logical and coherence layers of the standard. For other features and a comprehensive treatment, the interested reader is referred to the standard document [26]. Some of the standards extending or supporting the base SCI standard are referred to here as well.

1.3.1 Logical Layer

The logical layer specifies transaction types and protocols, packet types and formats, packet encodings, the standard node interface structure, bandwidth and queue allocation protocols, error processing, addressing and initialization issues, and SCI-specific CSRs.

Transactions. Transactions are split, consisting of a *request* and (for most transactions) of a *response subaction*. Correspondingly, the nodes involved are called the *requester* and the *responder*. A transaction is performed by sending a request packet from the requester to the responder and a response packet (if any) back to the requester. Packets carry addresses, command and status information, and data (depending on the type of transaction). Up to 64 transactions can be outstanding per node.

Each subaction consists of a *send* packet generated by the sender and an *echo* (acknowledgment) packet returned by the receiver; see Figure 1.2, packets (1) and (2), for an illustration of this *handshake* on a request subaction. The echo tells the sender whether the packet has been accepted (stored in the receiver's input queue for further processing) or rejected (e.g., due to a full input queue). In the former case, the sender can discard the send packet from its output queue (where it is required to be held); in the latter case, the sender retransmits the packet.

A consequence of this rejection/retry mechanism is that in-order delivery of packets cannot be guaranteed by the SCI hardware: a packet rejected by a busy queue can be overtaken by a later packet which happens to find space in the same queue. In addition, the retransmissions consume bandwidth and aggravate congestion on an already busy network or on the input queue of the "hot" receiver. Recent investigations have shown that this can have a significant impact on the sustained throughput of a heavily loaded SCI network; Chapter 6 reports on some of the results.

Figure 1.2 depicts the flow of packets for the more complicated case of a *remote* transaction which involves nodes sitting on different rings and one or more intermediate *agents*, e.g., bridges or switches coupling two or more SCI rings. An agent takes over responsibility for a subaction when it crosses rings, echoing the send packet on its source ring and forwarding it on to the target ring. An echo therefore is only a ring-local acknowledgement, i.e., a flow control action, not an end-to-end confirmation of the subaction. End-to-end confirmation of a transaction is only provided by a response.

Transaction types. Transactions fall into three categories: transactions with responses (*read*, *write*, and *lock* transactions), *move* transactions and *event* transactions. The transactions further differ in the amount of data they carry. An overview is given in Figure 1.3.

Transactions with responses are read, write, and lock accesses to DSM, in various flavors. Read transactions copy data from the responder to the requester, with 16, 64, or 256 bytes being transferred. In a 16-byte transaction,

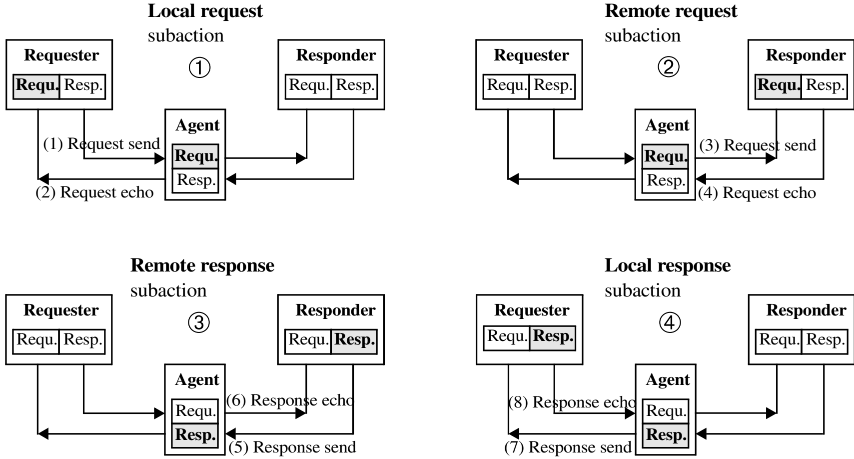


Fig. 1.2. Remote transaction phases

	Request		**Response**	
read*nn*	Header (16)		Header (16)	Data (*nn* = 0, 16, 64, or 256)
write*nn*	Header (16)	Data (*nn* = 16, 64, or 256)	Header (16)	
lock	Header (16)	Data (16)	Header (16)	Data (16)
move*nn*	Header (16)	Data (*nn* = 0, 16, 64, or 256)		
event*nn*	Header (16)	Data (*nn* = 0, 16, 64, or 256)		

Fig. 1.3. Transaction types (The numbers in parentheses denote the number of bytes of the packets. For space-accounting purposes, the length of the packet header includes the length of the trailer, the 2-byte CRC code.)

the size of the valid data may be between 1 and 16 bytes (selected-byte reads, *readsb*). 64 bytes, the size of a cache line in SCI, may be read in a coherent and a non-coherent manner, which distinguishes whether or not the data block is subject to coherence maintenance. Optionally, 256-byte transactions may be provided to transfer large data blocks in a non-coherent way. 0-byte reads are available as well, essentially only updating the coherence state of the addressed cache line. Besides data, responses to read transactions carry the status of the read operation.

Write transactions transfer data from the requester to the responder. The data sizes and the variants are defined as for read operations, with some minor exceptions; 0-byte writes are not supported. The responses to write requests acknowledge the success or signal the failure of the write. The 16-byte write transactions (selected-byte writes, *writesb*) are useful for manipulating control registers. The non-coherent 64-byte write operations can be employed for message passing according to the CSR standard.

The lock transaction specifies an indivisible operation (a *read–modify–write*) to be performed on the target memory location and the old memory contents to be delivered back to the requester. Such an atomic operation is required to build efficient mutual exclusion locks or semaphores in a shared memory system [24]. Early bus-based multiprocessor systems, for instance, supported the *test&set* operation by an indivisible bus and memory cycle. In a distributed system like an SCI network, the target node (the responder) must be able to perform the atomic read–modify–write operation. SCI defines a number of variants of this update operation to be provided by SCI memory controllers, among them the well-known *compare&swap* and *fetch&add* operations. The lock transaction carries data in both directions, the new (or update) value in the request packet and the old memory value in the response packet, plus status information.

Move transactions carry data from the source node to the destination node in a non-coherent way, like non-coherent writes. In contrast to write transactions, however, move operations do not have a response subaction. Consequently, moves are more efficient than writes, but correct data delivery is unconfirmed at the logical protocol layer. (There is still flow control due to the existence of the echo packet in the request subaction.) Moves are therefore expected to be used when reliable transfer is less important than timeliness, e.g., when data is written into a video frame buffer. Move transactions may address a single node (directed move, *dmove*) or multiple nodes (broadcast move). Broadcast moves and broadcast capability of nodes are specified as options only. Special protocols are defined that ensure reliable delivery of the broadcast even to nodes that are unable to accept it at first (because their receive queues are full).

Event transactions are even more special than moves in that they lack both confirmation and flow control, i.e., they have no response and do not generate an echo. Event transactions are to be accepted and delivered without

delays. Their intended use was to distribute a time stamp for global time synchronization in an SCI system; yet, they may also be used for fast data transfers under certain provisions [18].

Packets. Corresponding to the transaction phases, there are four basic types of SCI packets: *request send*, *request echo*, *response send*, and *response echo* packets. In addition, SCI specifies special packets like *init* and *sync* packets that are used during the system initialization process and for data stream re-synchronization purposes, respectively.

As an example, the format of the *request send* packet is shown in Figure 1.4. As illustrated, a packet consists of a contiguous sequence of 16-bit *symbols*. The packet header normally comprises seven symbols (14 bytes) and the trailer (CRC code) one symbol (2 bytes).

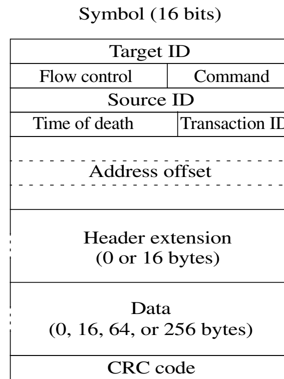


Fig. 1.4. Request send packet format (simplified)

The first symbol of the header contains the address of the destination node, *target ID*. Nodes receiving a packet on the incoming link inspect the target ID symbol to quickly determine whether to accept the packet (take it off the link) or to pass it on to the outgoing link.

The second symbol carries *flow control* information and the *transaction command*. The flow control field contains information for the ring bandwidth allocation and the queue allocation protocols, e.g., the packet priority and a field identifying whether and why the packet has been retransmitted. The command field specifies the type of the request transaction, e.g., readsb.

The third symbol provides the *source ID*, i.e., the address of the originator of the packet.

The fourth symbol specifies the *time of death* of the packet, i.e., the global time when it is to be discarded, and a 6-bit *transaction ID*. In conjunction with the source ID, the latter allows to distinguish between up to 64 outstanding transactions per node.

The following three symbols specify the *address offset* to be used by the responder, e.g., a memory address to fetch data from.

The optional *header extension* (16 bytes) is employed by certain cache coherency transactions only.

Depending on the request type, the packet carries 0, 16, 64, or optionally, 256 bytes of *data*.

The 16-bit CRC code consumes the final symbol of the packet. The CRC polynomial and a parallel hardware implementation model are specified in the SCI standard so that CRC codes can be computed and checked at full link speeds. The flow control information of the packet is excluded from the CRC calculation since it changes during the passage of the packet through the SCI network.

As compared to the request send packet depicted in Figure 1.4, response send packets carry status information in place of the address offset. Echo packets are considerably smaller than send packets, comprising four symbols (8 bytes) only; see, e.g., Chapter 5 for a description. The special packets required for network initialization and control are eight symbols (16 bytes) long. All packets therefore consist of an integer multiple of four symbols, which simplifies the design of systems that use wider data paths, whether as internal paths or external cabled links.

Packet encodings and idle symbols. The 16-bit symbols are the basic units for packet encoding. To encode and transmit a symbol, two extra signals are needed in addition to the 16 data signals: a *clock* signal that determines symbol boundaries and a *flag* signal that delineates the start and end of packets. The well-devised use of this flag ensures that there need not be special start and stop symbols locating packet boundaries.

A parallel SCI implementation, as exemplified by Dolphin's Link Controller chip [9], therefore typically comprises 18 parallel signal lines: the 16 data signals, the clock, and the flag. For serial implementation, the SCI standard encodes these 18 signals together with additional synchronization information in a 20-bit unit.

In order to enable SCI links to run continuously and at high speeds, the space between packets is filled up with so-called *idle symbols*. Idle symbols serve two purposes: (1) they allow SCI nodes to permanently synchronize the incoming data stream to the local clock, and (2) they transfer allocation and other network control information; see, e.g., Chapter 5 for details. At least one idle symbol must be present after a regular packet; special packets may be transmitted back to back. Coarsely speaking, idle symbols are created whenever a node takes a packet off the incoming link, and are replaced when a node has to send a packet. A node stripping off an idle symbol is responsible for saving and later re-inserting some of its control information such that, e.g., the allocation protocols can work properly.

Allocation protocols. The *queue allocation* protocols ensure that space will be reserved in the input queue of the receiver so that retransmitted packets

will eventually be accepted. On the other hand, the *bandwidth allocation* protocols guarantee that the sender will eventually obtain bandwidth to be able to send its packets. For a detailed discussion of these protocols and the information that is transmitted through an SCI network to facilitate their fair and efficient operation, refer to Chapter 5.

SCI node interface structure. An SCI node, more precisely its interface to the SCI network, must perform a number of challenging tasks. For instance, it must be able to insert packets onto the outgoing link, while concurrently stripping off packets addressed to itself from the incoming link or buffering (and later re-injecting) packets destined for other nodes, all at full link speeds. Moreover, it must participate in the allocation protocols as well as network maintenance and error processing, to that end observing, manipulating, and creating the control information in packets and idle symbols. It must keep track of outstanding transactions until confirmation or until a time-out, retransmit rejected packets, signal errors, and buffer transactions from or to the node's main components, e.g., the processor or a memory controller.

To address the complexity involved with these tasks, the SCI designers have proposed a standard interface of an SCI node, to be implemented in hardware. Figure 1.5 illustrates the interface model. Dolphin's Link Controller chip, for example, has been designed according to this model.

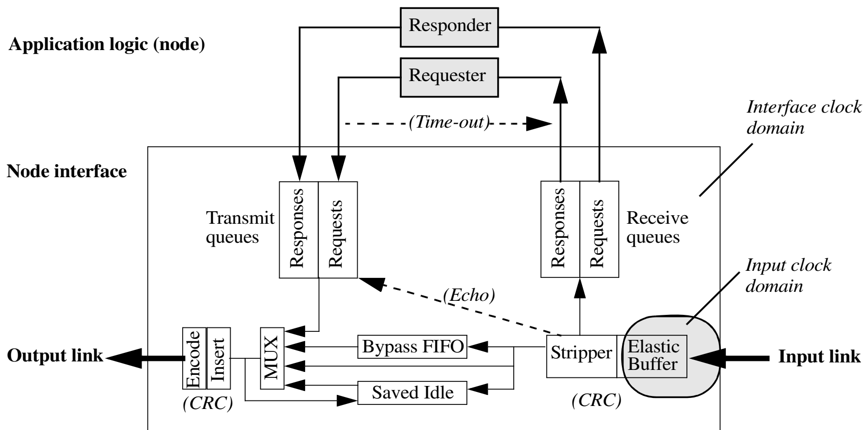


Fig. 1.5. Standard SCI node interface structure

The node interface has an incoming SCI link and an outgoing SCI link. On the input link, symbols arrive asynchronously w.r.t. the interface clock and carry their own clock signal. The first stage of the interface is therefore to synchronize the incoming data stream to the local clock domain in an asynchronous circuitry called the *elastic buffers*. Idle symbols are inserted or deleted in case the two clocks drift too far apart, subject to the rule that

packets remain intact. The rest of the node interface is strictly synchronous, greatly simplifying its design.

Packets addressed to the local node are taken off the link by the *stripper* circuit and appended to one of the receive queues for further processing by the node's main components (application logic). These are symbolized by a requester block (e.g., a processor) and a responder block (e.g., main memory). In case the packet is destined for another node, it is forwarded towards the outgoing link.

If, during the arrival of a packet, the local node is inserting a packet from a transmit queue onto the output link, the arriving packet must be buffered in order not to be lost. Storage for this purpose is called the *bypass FIFO*. A node is only allowed to inject one of its own packets onto the outgoing link whenever it has detected the bypass FIFO to be empty. Thus, the bypass FIFO must be designed to hold one packet of maximum size. As soon as the output link becomes available again, the bypass FIFO is emptied.

Incoming idle symbols are consumed whenever the node sends packets of its own. The buffer *saved idle* serves to store the arriving control and allocation information. Successive idles are merged into this buffer according to the fairness requirements of the allocation protocols. Idles are produced from this buffer whenever the receiver strips off a packet for the local node.

In general, the SCI node interface maintains two pairs of *queues*, serving as buffers until transmission bandwidth becomes available for outbound packets or until inbound packets can be processed by the node, respectively. SCI specifies separate queue storage for requests and responses. This is a simple and effective mechanism for eliminating a common cause of deadlock, by allowing responses to complete without regard for request congestion. The completion of responses eventually frees resources so that requests can be handled, ensuring forward progress.

A request or respond send packet needs to be held in its transmit queue until the corresponding echo arrives. When the echo arrives, the stripper signals this to the transmit queue. Depending on the type of the echo, the send packet is either discarded from the output queue (done echo) or needs to be re-sent (retry echo). Time-outs are used to deal with send packets whose echoes are overdue.

Error handling. SCI provides several features on the hardware and protocol levels to detect and isolate errors. Among them are: the CRC code and its calculation in hardware; hardware time-outs to detect damaged or lost packets; error status code fields in some packet types; the time of death that may be associated with a packet; distributed error logging according to the CSR Architecture; and the concept of CRC "stomping". The latter feature means that the CRC code of a packet is modified in a recognizable way if a problem during packet creation is encountered (and part of the packet is in transit already); an example of such a packet is a request echo that, for latency reasons, is generated during the receipt of a request send packet, which

eventually is detected to have an invalid CRC code. An error is to be logged whenever a packet is stomped which allows to isolate the source of the error.

Ringlets are required to have a so-called "scrubber" node which monitors ringlet activity and is responsible for network-maintenance functions, e.g., deleting corrupted or stale packets and idle symbols, handling packets with addressing errors, and circulating ring-maintenance information. A single scrubber node is identified and activated automatically during system initialization.

These features mainly contribute to detection, containment, and logging of errors. It is beyond the scope of the standard to specify how to handle and recover from errors. Software is expected to deal with these issues; see, e.g., Chapters 9 and 10.

1.3.2 Cache Coherence Layer

The cache coherence layer provides concepts and hardware protocols that allow processors to cache remote memory blocks while still maintaining coherence among multiple copies of the memory contents. Since an SCI network no longer has a central resource (i.e., a memory bus) that can be snooped by all attached processors to effect coherence actions, distributed directory-based solutions to the cache coherence problem had to be devised.

The resulting coherence protocols are quite complex. However, their implementation is *optional* only, not impairing system performance when they are not implemented or not being used. Further, *within* the cache coherence protocols, there are different sets of interoperable protocols: the *minimal* set, the *typical* set, and the *full* set. The implementor will choose one according to a cost-versus-performance trade-off.

The coherence protocols basically operate according to the multiple readers–single writer sharing model and use the write-invalidation scheme; no coherence model is prescribed by the standard.

At the core of the cache coherence protocols are the *distributed sharing lists* shown in Figure 1.6. Each shared block of memory has an associated distributed, doubly-linked sharing list of processors that hold a copy of the block in their local caches. The memory controller and the participating processors cooperatively and concurrently create and update a block's sharing list.

The coherence tags and related information required for coherence maintenance are depicted in Figure 1.6 as well. With the standard size of an SCI memory/cache block (64 bytes), this results in only a few percent storage overhead in the main memory and caches. The overhead is *fixed*, i.e., independent of the system size, which is an important building block of the scalability of the coherence protocols. Note that cache coherence can be enabled or disabled on portions of the memory, typically on a per-page basis. Memory blocks can therefore be cached in a coherent or non-coherent manner, or be uncached at all.

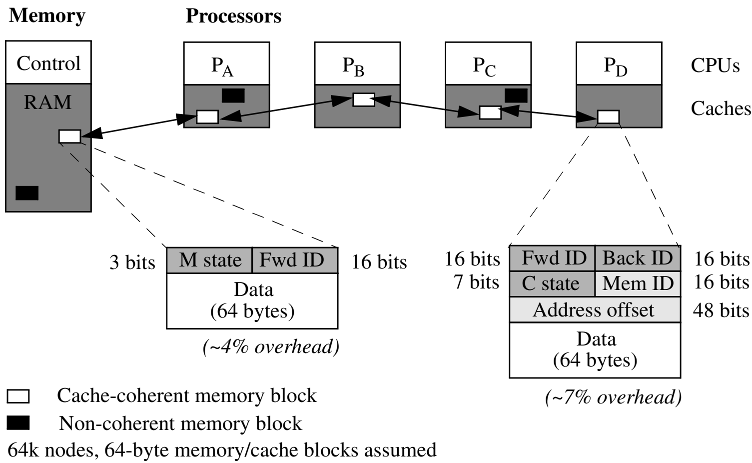


Fig. 1.6. Sharing list and coherence tags of SCI cache coherence protocols

Each shared, coherent memory block has a 3-bit *memory state* field and a 16-bit *forward pointer* that holds the SCI node ID of the node at the head of the sharing list. The cache entries contain a 7-bit *cache state* field, a *backward pointer* and a *forward pointer* (node IDs) for list maintenance, and the full address of the cached data block, consisting of the SCI node address of the memory module, *memory ID*, and the *address offset* within that module.

It is beyond the scope of this introduction to describe in detail the states and state transitions of the memory and cache blocks, the coherent transactions and the actions and issues involved with updating the sharing lists. Only the basic principles are given in the sequel.

When a processor accesses a cacheable, coherent memory block, e.g., by a coherent-read transaction, the memory controller updates the state of the block and saves the address of the requesting processor in its forward pointer field.

If no cached copies of the block exist at that time (i.e., the sharing list is empty), the requester becomes the head of a new list; the memory ships the data block and the requesting node initializes its backward pointer, cache state, and address fields.

If a sharing list for the block already exists at the time of the request, the requesting node will become head of the sharing list as follows. The memory controller updates its memory state and the forward pointer to point to the requester (the designated head); it returns the pointer to the current head back to the requester. Note that the memory does not ship the data block to the requester since it may have a stale copy of the data only. The requester uses this information to update its coherence tags and pointers to become the new head (provisionally); it further sends a transaction to the old head to identify itself as the new head and, simultaneously, request the data block.

The old head updates its backward pointer, thereby degrading itself to a regular list entry, modifies its cache state, and returns the (up-to-date) data block from its cache to the requester. The requester finally loads the data into its cache and definitively establishes itself as the new list head.

When a processor requires write access to a shared data block, it first makes itself the new head of the block's sharing list, since only the head of the list has write permission. It then deletes all other entries down the list in order to obtain an exclusive copy of the block; in other words, all other processors have to discard their copies of the block from their caches. Depending on when the node at the head of the list is actually permitted to modify the data, different coherence models can be realized; under a sequential consistency regime [24], the node would have to wait until purging the rest of the list is completed, while a more relaxed consistency model would allow the head to proceed before completion of the deletion process.

Notice that the list insertion sequence described above involves three nodes, two transactions with responses (requester $\rightleftharpoons$ memory and requester $\rightleftharpoons$ old head), and transitory states to prepend the requesting node to the sharing list. Furthermore, other nodes may concurrently request access to the same memory block and must be added to the block's sharing list as well; the order of the list insertions is defined by the arrival times of the requests at the memory controller.

Despite the distributed nature of the protocol, both in terms of space and time, and the concurrency issues involved, list insertions (and deletions as well) must appear atomically. This requirement is further exacerbated by the fact that transactions (or subactions thereof) may fail, potentially leaving nodes or sharing lists in inconsistent states unless provisions for recovery are defined. The SCI coherence protocols use implicit conditional atomic transactions, like *compare&swap*, and never require any part of the directory structure to be locked, in contrast to the usual software protocols for maintaining a coherence directory.

While the above discussion disclosed only a small part of coherence maintenance actions, it should have become clear that the designers of SCI had to deal with many difficult and subtle correctness and robustness problems pertaining to the cache coherence protocols. This explains the complexity of the coherence specification. Simulations of the specification (given by the C code) have greatly helped in designing and debugging the protocols. Despite the well-devised protocols and the C code, implementation of SCI cache coherence, and especially interfacing it to typical processor coherence schemes, remains a major challenge, though; see, e.g., Section 1.4.2.

The more advanced coherence protocol sets introduce optional optimizations for important sharing or access conditions. A good example is *pairwise sharing* where, e.g., one producer and one consumer share a data block; in this case, data may be transferred directly from the producer's cache to the consumer's cache, without manipulating the sharing list on each modification and

without involving memory. A further example is *queued-on-lock-bit (QOLB) synchronization* which provides FIFO access to an exclusive resource; the sharing list structure is exploited to hand over the resource without needless communication.

1.3.3 Extensions

A number of projects have been started to develop standards that extend or support SCI in a *compatible* way. The rationale for these projects was mainly to specify issues and mechanisms that were regarded to be important, but could not be covered by the base standard anymore, or that could be postponed until the basic concepts were finished.

Some of these projects have successfully produced approved standards, among them:

– *Low Voltage Differential Signals (LVDS) for SCI* (IEEE Std 1596.3-1996) [27] that defines low-voltage differential signals (250 mV swing) and signal encodings for data paths that are 4, 8, 32, 64, or 128 bits wide; and
– *Shared-Data Formats Optimized for SCI* (IEEE Std 1596.5-1993) [28] which defines formats for data transfers among heterogeneous computing systems in a distributed environment based on SCI.

The nearly completed standards project *Parallel Links for SCI* (IEEE P1596.8) specifies the cables and connectors that have become favored by SCI users. Another successful standardization effort seems to be the *SCI Physical Layer API* project (IEEE P1596.9) which specifies a general-purpose shared-memory hardware abstraction layer designed for SCI (but usable on other shared-memory systems as well). This upcoming standard describes functionality required to set up and run DSM systems with little overhead; an in-depth discussion is given in Chapter 10.

A standardization project that has been active for a long time is the attempt to specify *Cache Optimizations for Large Numbers of Processors using SCI*, most often called *KiloProcessor Extensions* (IEEE P1596.2). This project has been motivated by the observation that, due to the linear structure of the sharing lists, SCI cache coherence actions will exhibit poor performance when a large number (perhaps thousands) of nodes share a single memory/cache block. Further, congestion can arise in the network or at the memory controller when many nodes concurrently access a given data block. The KiloProcessor Extensions address this deficiency by providing tree-like sharing structures compatible with the linear sharing lists of SCI, and by supporting more efficient mechanisms for distribution of widely shared data and for purging copies, e.g., by using write-update mechanisms and combinable operations when possible. The aim is to reduce data-access latencies to an order logarithmic in the number of participating processors rather than linear. A description of the concepts under discussion can be accessed at [35].

A highly promising extension of the SCI standard was the *High Speed Memory Interface (SyncLink)* standardization effort (IEEE P1596.7-199X), until recently driven by SLDRAM, Inc., a non-profit corporation sponsored by most of the DRAM manufacturers. The SyncLink project specified a high-bandwidth, synchronous-link interface optimized for interchanging data between a memory controller and DRAM chips. The mechanisms are described in a draft standard [29]. Although the technology did work, the companies supporting SLDRAM have withdrawn recently. The reasons and the future prospects of the technology are briefly addressed in Chapter 27.

1.4 Applications of SCI

SCI was originally conceived as a shared-memory interconnect and a first implementation of such a network emerged in 1994 (in the HP/Convex Exemplar SPP multiprocessor). However, SCI's flexibility and performance potential also for other applications, e.g., as a system area network, was soon realized and leveraged by industry. This section introduces several of these "classical" applications of SCI and provides examples of commercial systems exploiting SCI technology. Upcoming applications and more recent developments, e.g., SyncLink [29] and SerialPlus [30], are covered in Chapter 27.

1.4.1 System Area Network for Clusters

Compute clusters, i.e., networks of commodity workstations or PCs, are becoming ever more important as cost-effective parallel and distributed computing facilities. An SCI system area network can provide high-performance communication capabilities for such a cluster. In this application, the SCI interconnect is attached to the I/O bus of the nodes (e.g., PCI) by a peripheral adapter card, very similar to a LAN; see Figure 1.7. In contrast to a LAN though (and most other system area networks as well), the SCI cluster network, by virtue of the common SCI address space and associated transactions, provides hardware-based, *physical* distributed shared memory. Figure 1.7 shows a high-level view of the DSM. An SCI cluster thus is more tightly coupled than a LAN-based cluster, exhibiting the characteristics of a NUMA (non-uniform memory access) parallel machine.

The SCI adapter cards, together with the SCI driver software, establish the DSM as depicted in Figure 1.8. (This description pertains to the solutions taken by the Dolphin SBus-SCI and PCI-SCI adapter cards; see Chapter 3.) A node that is willing to share memory with other nodes (e.g., A), creates shared memory segments in its physical memory and exports them to the SCI network (i.e., SCI address space). Other nodes (e.g., B) import these DSM segments into their I/O address space. Using on-board address translation tables (ATTs), the SCI adapters maintain the mappings between their local

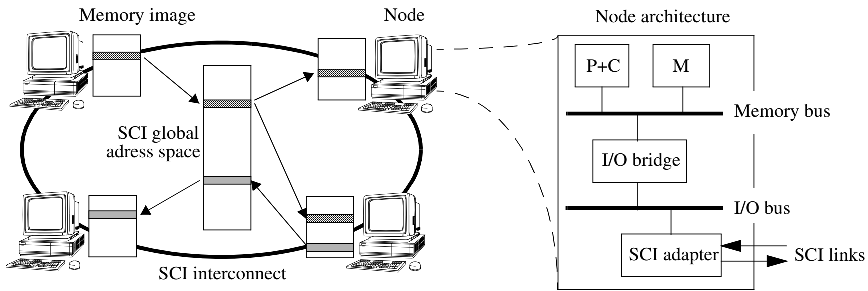


Fig. 1.7. SCI cluster model

I/O addresses and the global SCI addresses. Processes on the nodes (e.g., i and j) may further map DSM segments into their virtual address spaces. The latter mappings are conventionally being maintained by the processors' MMUs.

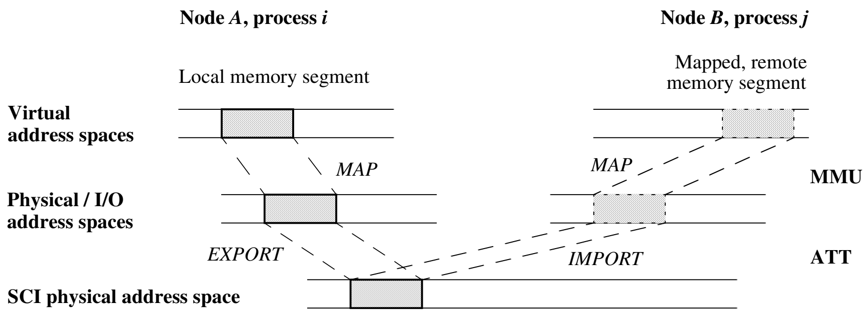


Fig. 1.8. Address spaces and address translations in SCI clusters

Once the mappings have been set up, internode communication may be performed by the participating processes at *user level*, by simple CPU load and store operations into DSM segments mapped from remote memories. The SCI adapters translate I/O bus transactions that result from such memory accesses into SCI transactions, and vice versa, and perform them on behalf of the requesting processor. Thus, remote memory accesses are both transparent to the requesting processes and do not need intervention by the operating system. In other words, no protocol stack is involved in remote memory accesses, resulting in low communication latencies even for user-level software.

Currently there is only one commercial implementation of such an SCI cluster network, the SBus-SCI and PCI-SCI adapter cards (and associated switches) offered by Dolphin Interconnect Solutions; Chapter 3 describes the adapter cards in detail. These cluster products are used by other companies to build key-turn cluster platforms; examples are Sun Microsystems, offering

high-performance, high-availability server clusters [10], and Scali Computers and Siemens providing clusters for parallel computing based on MPI (see Chapters 8 and 14 as well as [37]). The research and development projects and products described in this book are to a large extent based on Dolphin's SCI cluster technology.

In addition, there are two research implementations of PCI-SCI adapter cards, one developed at Technische Universität München and described in Chapter 4, the other one developed at CERN as a prototype to investigate SCI's feasibility and performance for demanding data acquisition systems in high-energy physics [2].

The description of an SCI cluster interconnect given above does not address many of the low-level problems and functionality that the implementation has to cover. For instance, issues like the translation of 32-bit node addresses to 64-bit SCI addresses and vice versa, the choice of the shared segment size, error detection and handling, high-performance data transfers between node hardware and SCI adapter, and the design and implementation of low-level software (SCI drivers) represent a spectrum of research and development problems. Parts II and IV of this book describe these problems and corresponding solutions in some detail. In addition, there is a wide design space for SCI interconnection networks which is explored by some of the chapters in Part III.

As an example of specific functionality and characteristics of SCI cluster hardware, consider Dolphin's recent PCI-SCI adapter cards. There are two ways to transfer data to and from the SCI network. The first method works as described above: a node's CPU actively reads data from (writes data to) a remote memory using load (store) operations into a DSM address window mapped from the remote node; this can be fully done on user level, resulting in round-trip latencies as low as 4–5 μs, but occupying the CPU for moving the data. The second method involves a direct memory access (DMA) engine on the SCI adapter that copies the data into and out of the node's memory; while this approach relieves the CPU, it has higher startup costs since the SCI driver software has to be involved to set up the DMA transfer. The SCI adapters provide several features optimizing the performance of the data transfers into and out of the node, among them using PCI burst operations, combining consecutive small SCI transactions addressing a contiguous memory region into a larger transaction, and prefetching. Under ideal circumstances (see Chapter 2), throughput of more than 80 MByte/s into and out of a node can be achieved, limited by the memory and I/O architecture of the node. The SCI interface chips and links can provide a bandwidth of 500 MByte/s.

An important property of such SCI cluster interconnect adapters is worth pointing out here. Since an SCI cluster adapter attaches to the I/O bus of a node, it cannot directly observe, and participate in, the traffic on the memory bus of the node. This therefore precludes caching and coherence

maintenance of memory regions mapped to the SCI address space. In other words, remote memory contents are basically treated as non-cacheable and are always accessed remotely. Current SCI cluster interconnect hardware does not implement cache coherence capabilities therefore. Note that this property raises a performance concern: remote accesses (round-trip operations such as reads) must be used judiciously since they are still an order of magnitude more expensive than local memory accesses.

The basic approach to deal with the latter problem is to avoid remote operations that are inherently round-trip, i.e., reads, as far as possible. Rather, remote writes are used which are typically buffered by the SCI adapter and therefore, from the point of view of the processor issuing the write, experience latencies in the range of local accesses, several times faster than remote read operations. In Part V of the book, several chapters describe how considerations like this influence the design and implementation of efficient message-passing libraries on top of SCI.

Several chapters in Part VI deal with how to overcome the limitations of non-coherent SCI cluster hardware, in particular for the implementation of shared-memory and shared-object programming models. Techniques from software DSM systems, e.g., replication and software coherence maintenance techniques, are applied to provide a more convenient abstraction, e.g., a common *virtual* address space spanning all the nodes in the SCI cluster.

1.4.2 Memory Interconnect for Cache-Coherent Multiprocessors

The use of SCI as a cache-coherent memory interconnect allows nodes to be even more tightly coupled than in a non-coherent cluster. This application requires SCI to be attached to the memory bus of a node, as shown in Figure 1.9. At this attachment point, SCI can participate in and "export", if necessary, the memory and cache coherence traffic on the bus and make the node's memory visible and accessible to other nodes. The nodes' memory address ranges (and the address mappings of processes) can be laid out to span a global (virtual) address space, giving processes transparent and coherent access to memory anywhere in the system. Typically, this approach is adopted to connect multiple bus-based commodity SMPs to form a large-scale, cache-coherent (CC) shared-memory system, often termed a CC-NUMA machine.

There are three notable examples of SCI-based CC-NUMA machines: the HP/Convex Exemplar series, now in its second generation [6][40], the Sequent NUMA-Q multiprocessor [32][7], and the Data General AViiON scalable servers [5]. The latter two systems comprise bus-based SMP nodes with Intel processors, while the Exemplar uses HP PA-RISC processors and a non-blocking crossbar switch within the nodes. The inter-node memory interconnects are proprietary implementations of the SCI standard, with specific adaptations and optimizations incorporated to ease implementation and integration with the node architecture and to foster overall performance.

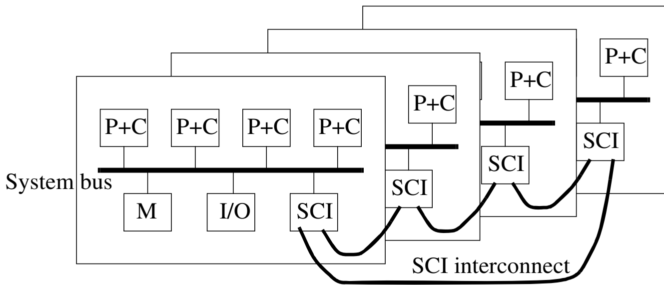


Fig. 1.9. SCI-based CC-NUMA multiprocessor model

The major challenge in building a CC-NUMA machine is to bridge the cache coherence mechanisms on the intra-node interconnect (e.g., the SMP bus running a snooping protocol) and the inter-node network (SCI). Since the *Sequent NUMA-Q* machine is well documented, it is used as a case study to illustrate the essential issues in building such a bridge and making the protocols interact correctly. A block diagram of the adapter board coupling the node bus and the SCI network is given in Figure 1.10; Sequent's SCI implementation is called *IQ-Link* [33].

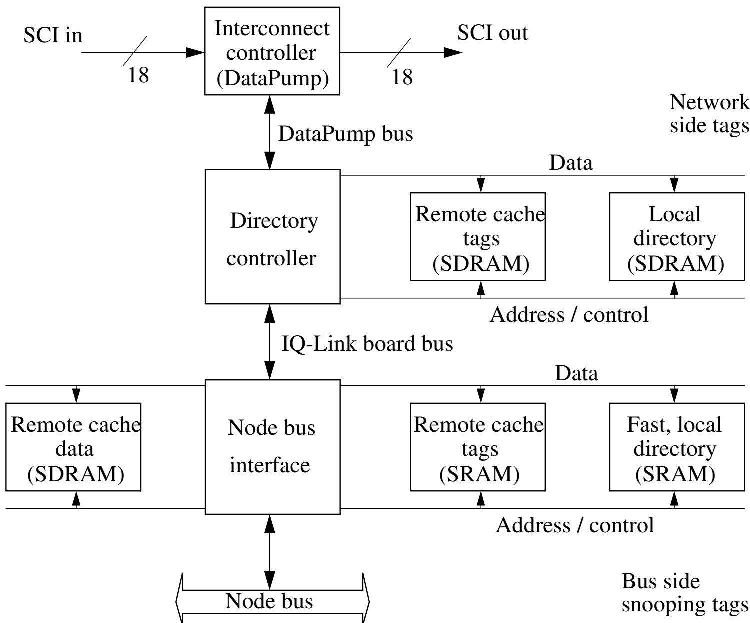


Fig. 1.10. Block diagram of Sequent's IQ-Link board

The NUMA-Q machine interconnects commodity bus-based SMP nodes by a ring structure, with typically four Intel Pentium Pro or Xeon processors per node. Every processor in the system has a common view of the system-wide memory and I/O address space. The IQ-Link board plugs into the node bus and participates in the standard MESI bus snooping protocol on behalf of remote nodes.

The essential building block of the interface is the *remote cache*. This is a large (32 MByte in first-generation systems [32]), expandable, 4-way set-associative cache, maintaining copies of blocks that are fetched from remote memories and representing the node to the rest of the system. The remote cache is kept coherent with local processor caches by means of the bus-snooping protocol, and with processor caches on remote nodes and remote memories by SCI's distributed directory protocol. The SCI directory protocol only operates on the remote cache; it is unaware of the local processor caches. Vice versa, the local caches are not aware of other nodes; they are supplied with remote data from the remote cache by virtue of the snooping protocol.

The block size of the remote cache is 64 bytes. The remote cache tags carry information as shown in principle in Figure 1.6. The implementation deviates from the standard, though. Forward and backward pointers are only 6 bits wide (allowing 64 nodes in a first-generation system); only an address tag of 13 bits is held, rather than a full 48-bit address offset. Notice that the remote cache tags are duplicated so that the two protocols may operate concurrently on the remote cache. Since only the SCI protocol is aware of other nodes, only the network-side copy of the coherence tags need to have pointers to other nodes; the bus-side tags only contain address and state information. The bus-side tags must be accessible at bus speed and are therefore implemented in SRAMs.

The *local directory* maintains information about the state of the blocks that have their home in the local memory; see the memory tags in Figure 1.6. Again, the implementation differs from the standard, providing a 2-bit memory state and a 6-bit list head pointer only. The local directory tags are duplicated as well and again need not contain pointers to other nodes on the bus side.

The *node bus interface* is an agent that snoops the node bus and manages the bus-side tags as well as the remote cache data arrays. When the bus interface controller detects that a bus request to a remote memory can be satisfied by the remote cache on the IQ-Link board, it supplies the requested block on the bus, with a latency close to that of a local memory access. Accesses that cannot be satisfied locally are forwarded to the directory controller. Further, the bus interface forwards incoming requests that access data, change state, or invalidate blocks, to the node bus on behalf of the directory controller. The bus interface can handle multiple outstanding requests and operate on multiple incoming requests.

The *directory controller* manages the network-side tags and executes the SCI coherence protocol. In first-generation systems, this component is implemented as a programmable protocol processor. The reason behind this decision is to avoid the risks and costs associated with a hardware implementation of the complex SCI cache coherence protocols; a firmware implementation is more easily debugged and optimized than hardware. Clearly, this does have performance impacts, as indicated below. The protocol processor can handle up to 12 protocol transactions and one interrupt transaction simultaneously by invoking a firmware "task" for each operation and supporting fast hardware task switch. Special logic for bit-field processing is available.

The *interconnect controller* is an SCI interface as depicted in Figure 1.5, implementing the physical and logical layer SCI protocols. The Vitesse DataPump chip is employed for NUMA-Q machines, driving 1 GByte/s, 18-bit wide SCI links.

Several problems arise due to two cache coherence protocols interacting in the machine [7]. An obvious problem is the different cache line size on the node bus (32 bytes) and on SCI (64 bytes). Some limitations of the Pentium Pro bus (in first-generation systems) become difficult to work around when long-latency operations appear. For example, since the bus expects in-order responses to requests, a memory access that must be satisfied remotely must be signaled to the bus as delayed by the IQ-Link bus interface controller. When the reply comes back eventually, the bus interface must place it on the bus and complete the deferred transaction. A delayed reply does not automatically update main memory, requiring special actions by the IQ-Link interface in case a locally allocated block is concerned. Furthermore, when two read-exclusive requests appear on the bus back-to-back, the second one is aborted by the bus and must be retried later on, rather than being buffered to implement a reply more efficiently. Subtle issues also arise with serialization of concurrent remote and local accesses to a memory block. This is done in SCI at the home memory of the block. When the home is an SMP node, the bus protocol must be involved on the home node to update the states of local copies of the block on remote accesses. In fact, the bus becomes the serialization point rather than the IQ-Link agent.

Performance characteristics of the NUMA-Q machine are given in [7]. Initial systems can sustain 30 MByte/s data transfer rate in each direction through the IQ-Link board. Remote memory access latencies are about 3 μs minimum and up to 9 μs under heavy load; more than 60% of these remote latencies are typically spent traversing the IQ-Link boards. Optimized microcode and hardware assist in the directory controller were identified as promising techniques for reducing these latencies. Nevertheless, the NUMA-Q machine delivers good performance on the commercial workloads it is designed to handle [32].

1.4.3 I/O Subsystem Interconnect

SCI can be used to connect one or more I/O subsystems to a computing system in novel ways. The shared SCI address space can include the I/O nodes which then are enabled to directly transfer data between the peripheral devices (in most cases, disks) and the compute nodes' memories using DMA; software needs not be involved in the actual transfer. Remote peripheral devices in a cluster, for instance, thus can become accessible like local devices, resulting in an I/O model similar to SMPs; remote interrupt capability can also be provided via SCI. High bandwidth and low latency, in addition to the direct remote memory access capability, make SCI an interesting candidate for an I/O network.

There are currently two commercial implementations of SCI-based I/O system networks. One is the GigaRing channel from SGI/Cray [36], the other one the external I/O subsystem interconnect of the Siemens RM600 Enterprise Servers, based on Dolphin's cluster technology [38].

The *SGI/Cray GigaRing channel* is a high-speed I/O network using point-to-point links organized as two counter-rotating rings. It is used to interconnect peripheral controllers, network bridges, and SGI/Cray supercomputers in a variety of configurations. For example, a number of I/O nodes attached to a single GigaRing can form the I/O subsystem of an SGI/Cray computing system. One or multiple GigaRing channels with peripheral devices can be shared by several, even heterogeneous SGI/Cray systems, e.g., a T90 vector machine and a T3E massively parallel processor. The GigaRing therefore represents a common I/O system interconnect for the various types of SGI/Cray computers.

The GigaRing channel heavily borrows from the SCI concepts and protocols, but deviates from the standard in many respects. Unused features have been discarded and new ones added to meet the requirements of a high-speed I/O channel.

The physical layer of the GigaRing is implemented by the proprietary GigaRing node chip. This chip is basically a simplified SCI node interface, but with 32-bit links in both directions and a 64-bit interface to the client (compute or I/O node). The chip and the wide links provide more than 1 GByte/s bandwidth per direction.

The logical layer uses SCI's ring access, bandwidth allocation, and queue reservation protocols with minor modifications. However, the packet types are tailored specifically to the requirements of I/O, providing transactions for small-message transfers, DMA operations, and maintenance and control. Packet formats differ significantly from the standard, with 32-bit symbols, 16-byte headers, up to 256 bytes of payload, and a 4-byte trailer. GigaRing also provides mechanisms for end-to-end congestion control, an area which is not covered by the SCI standard. Obviously, the cache coherence protocol is not implemented on GigaRing.

The two counter-rotating rings provide for increased availability: a broken ring can be disabled (ring masking), and the two rings can be appropriately cross-connected to isolate a broken node, resulting in one intact ring (ring folding). Disabling and reconfiguration can be effected by maintenance packets sent on the GigaRing channel.

Siemens provides an *external I/O expansion* for the RM600 Enterprise Servers, using Dolphin's PCI-SCI cluster components. SCI rings can be used to couple multiple PCI controllers such that they appear as one large PCI bus with more than 100 PCI slots. SCI can also be used to directly couple nodes in a cluster configuration.

1.4.4 Large-Scale Data Acquisition System

A special form of an I/O system is a data acquisition system. The most challenging data acquisition tasks exist in high-energy physics applications, e.g., particle detectors or nuclear fusion experiments. As an example, consider the ATLAS experiment at the Large Hadron Collider (LHC), now under construction at CERN. It is expected that merely the second-level data acquisition and real-time selection system will comprise about 2000 data sources and 1000 processing nodes. These will have to be connected by a high-performance network that must be able to sustain a throughput of several GByte/s. See Chapter 23 for a detailed explanation of the structure and requirements of this data acquisition system.

It is not surprising, therefore, that researcher have investigated SCI as the basis for these interconnects for many years. CERN, together with associated institutions, have designed and demonstrated SCI components, network structures, and software in various projects. Early research is documented in [2], and a current project to build an SCI prototype system is described in Chapter 23. Another contribution in this book, Chapter 6, studies topologies and performance characteristics of SCI networks for plasma fusion devices, currently using simulation as the main tool.

1.5 Related Communication Networks and Concepts

Besides SCI, a number of related communication networks and concepts have been proposed for building clusters or tightly-coupled multiprocessors (NUMA or CC-NUMA systems, respectively). For the sake of brevity, only the most significant differences between these related interconnects and the SCI standard and SCI implementations are pointed out in the sequel.

Myricom's *Myrinet* [1] is a high-speed system area and local area network that has its origins in the interconnect technology of a massively parallel machine. Network interface cards attaching to workstations' I/O buses, high-speed links, switches, and a wealth of software, predominantly optimized

message-passing libraries, are available to facilitate the construction of high-performance compute clusters. In contrast to SCI, a shared address space across the nodes in a cluster is not provided by the technology. However, the adapter card hosts a programmable processor, which allows specific communication mechanisms to be implemented, among them abstractions akin to DSM [11]. Chapter 2 describes Myrinet in more detail and investigates its performance by several communication benchmarks.

Other cluster interconnects supporting high-bandwidth, low-latency message-passing communication include *ParaStation* [43] and *ATOLL* [4]. *PAPERS* is an interconnect technology focussing on fast aggregate communication and computation, e.g., barrier synchronizations and global reduction operations [21]. The *Gigabyte System Network* (GSN) is an implementation of the ANSI standard developed under the name HIPPI-6400; it is currently being offered by SGI for highest-throughput (close to 800 MByte/s) cluster computing and as a storage area network [39].

The Compaq/Digital *Memory Channel* (MC) [14][15] network is conceptually similar to SCI in that it provides a hardware-based, non-coherent physical DSM in a cluster of workstations or SMPs; low communication latencies and overheads are of primary concern as well. However, the MC uses the reflective memory concept which mirrors write operations to a memory in other, connected memories. The nodes' memories can be connected to each other by address mappings similar to SCI, with page-level granularity. The network adapters attach to the I/O bus and can be accessed from user level, as in SCI. Only writes to remote memories are facilitated, though, no read accesses. The most important difference to SCI is that the shared address space of MC is provided by a separate device, the MC Hub, which imposes a strict limit on the number of nodes that can be attached. MC provides in-order delivery of writes, flow control in hardware, and more sophisticated error-detection and reliability features than SCI.

The *SHRIMP* multicomputer [3] introduced the notion of Virtual Memory Mapped Communication (VMMC). The concept is similar to SCI DSM in that it allows applications to transfer data directly between two virtual memory address spaces over the network. The basis are virtual memory-mapped network interfaces and import-export mappings similar to SCI. Two transfer strategies are supported: deliberate update, which is an explicit transfer (send) of data, and automatic update, which reflects operations on exported local memory segments in the remote memory (by hardware means). A number of communication libraries have been implemented exploiting the advantages of VMMC, most importantly user-level access to the network and the opportunity to perform zero-copy data transfers [8]. In contrast to SCI, SHRIMP does not provide a cluster-wide shared address space. The first implementation was based on a proprietary Intel routing backplane as used in the Paragon multiprocessor; the second implementation adapts Myrinet to support the VMMC concept [11].

ServerNet, developed by Compaq/Tandem [22][23], is a flexible system area network that can support communication among processors (by memory-to-memory data transfers), I/O traffic between processors and peripheral devices, and even between I/O controllers. The ServerNet interconnect has interfaces to processors and associated memories, and to peripheral buses, with the goal to unify inter-processor and I/O connectivity. Processors and memories can be connected to two redundant, multi-stage interconnect fabrics. Similar to SCI, ServerNet supports both read and write transactions directed towards remote memories or I/O nodes in order to pull or push data across the network without software intervention at the remote node; remote interrupts are supported as well. The network hardware provides delivery and ordering guarantees as well as fault detection and isolation features at various architectural levels. ServerNet is a proprietary implementation, originally designed to build high-throughput, reliable I/O systems [23] and fault-tolerant clusters. More recently, however, ServerNet-II was proposed as a native implementation of the Virtual Interface (VI) Architecture standard [12], targeted also towards high-performance compute clusters for scientific applications [19].

The *Virtual Interface (VI) Architecture* industry standard [12] summarizes many of the concepts and approaches developed in earlier projects, like U-Net [42] or SHRIMP [3], for high-throughput and low-latency communication in clusters. VI Architecture specifies cluster communication concepts and a network interface architecture in an open, implementation-independent fashion. Notable concepts are: virtualization of the network interface to user processes; organization of send and receive queues and descriptors; protected, user-level access to the network; registration and use of portions of the virtual memory for communication; virtual-to-physical address translation for these memory regions, bypassing the operating system; avoidance of intermediate data copies as well as of interrupts and context switches, in order to minimize communication latency and CPU overheads; and remote memory accesses without involving software on the remote node. It is expected that these concepts will have a significant influence on future products designed for fast cluster communication.

As to scalable cache-coherent multiprocessors, the SGI *Origin 2000* [31] seems to be the only commercial alternative to the SCI-based CC-NUMA machines mentioned in Section 1.4.2. The Origin 2000 is an advanced, optimized server machine with origins in the Stanford DASH project. The nodes in the Origin are equipped with two MIPS processor which are connected to each other, to local memory, and to I/O and network interfaces by a single hub chip. Up to 512 nodes can be interconnected in a hypercube-like topology. Both the memory in the system and the I/O address space are globally addressable. Cache coherence is maintained by a bit-vector directory scheme where up to 64 bits per memory block keep track of where copies of the block exist. The bit vector can be adapted to identify a single node per bit (full

bit vector) or, for systems larger than 128 processors, a group of processors (coarse bit vector). The directory memory of a node is associated with its local memory and grows proportionally as the latter is extended. The hardware and the operating system have mechanisms to count the references to memory pages and migrate the page to the busiest requester, if this promises to reduce the average memory access latency.

S3.mp was a technology development project by Sun [34] that aimed at connecting off-the-shelf workstations (SparcStations) into a large-scale, cache-coherent multiprocessor. The project developed a memory controller that attaches to the Mbus of a SparcStation, handles accesses to local and remote memories, and executes the cache coherence protocol by two microcoded protocol engines. S3.mp uses an invalidation-based protocol which keeps track of nodes sharing a memory block using singly-linked lists. The second major contribution of the project was the interconnection network, comprising an interconnect controller with an integrated arbitrary topology router and high-speed serial fiber-optic links. Unlike the SCI-based CC-NUMA multiprocessors which use special packaging, S3.mp nodes were anticipated to be spatially distributed (up to 200 m) and to be connected in arbitrary topologies.

Finally, the *Wisconsin Wind Tunnel Project* [20] pursues in-depth research on the design trade-offs for cost-effective DSM parallel machines, taking further many of the issues that SCI addresses. Research areas include network interface design, cooperation of hardware and software to manage the DSM, combining shared memory and message passing for parallel computation, and the use of system-wide prediction and speculation techniques, e.g., to accelerate coherence actions.

1.6 Concluding Remarks

SCI is an interconnect standard that specifies leading-edge high-speed networking and distributed shared memory (DSM) technology. Although the concepts and protocols are well devised and were published as an open specification in the early 1990s already, SCI has not lived up to its promise of becoming an "open distributed bus". Chapter 27 presents an insider's view on the obstacles to rapid and wide-spread acceptance by industry.

SCI was, however, rather quietly adopted by several companies that recognized its superior concepts and protocols as well as its potential of high performance, but had no interest in implementing and providing SCI as an open interconnect. A number of proprietary implementations and products therefore appeared over the years, ranging from high-performance cluster interconnects, to shared-memory multiprocessor networks with cache coherence implemented in hardware, and high-speed I/O subsystem interconnects. In particular, the CC-NUMA machines based on SCI technology

from HP/Convex, Sequent, and Data General turned out to be quite successful.

Adoption of workstation clusters using an SCI interconnect (and its DSM), is however slower than expected, despite the superior performance characteristics of SCI cluster networks available today. The reasons may well be that for many years there has been only one serious vendor of SCI adapters and switches, Dolphin Interconnect Solutions, and that progress on developing software for the efficient use of these clusters has been slow. This even holds for message-passing libraries and more so for shared-memory programming paradigms that may take advantage of SCI's DSM. This book is intended to contribute to faster and wider acceptance of SCI clusters in academia and industry by summarizing the state of the art of SCI cluster computing, pointing out achievements and remaining obstacles as an aid for potential users.

SCI's development and influence is not finished yet. For example, recent interconnect standard projects like SyncLink, a high-speed memory interface, and Serial Express (now tentatively named SerialPlus), an extension to the SerialBus (IEEE 1394) interconnect, directly emerged from SCI or are heavily influenced by SCI concepts. SCI also plays a role in the debate on future I/O systems (NGIO versus FutureIO). These current developments and future directions are explored in more detail in Chapter27.

References

1. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–36, Feb. 1995.
2. A. Bogaerts et al. *RD 24 Status Report: Application of the Scalable Coherent Interface to Data Acquisition at LHC*. Oct. 1996. http://nicewww.cern.ch/~hmuller/~HMULLER/docs/report96.pdf.
3. M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, pages 21–28, Feb. 1995.
4. U. Bruening, L. Schaelicke. ATOLL: A High-Performance Communication Device for Parallel Systems. *Proc. Advances in Parallel and Distributed Computing.* Shanghai, 1997.
5. R. Clark. *SCI Interconnect Chipset and Adapter: Building Large Scale Enterprise Servers with Pentium II Xeon SHV Nodes.* White Paper. Data General Corp. 1999. http://www.dg.com/about/html/sci_interconnect_chipset_and_a.html.
6. Convex Computer Corp. *Convex Exemplar Architecture.* Technical Document DHW-014. Convex Computer Corp., Nov. 1994.
7. D. E. Culler, J P. Singh, with A. Gupta. *Parallel Computer Architecture: A Hardware-Software Approach.* Morgan Kaufmann 1998.
8. S. N. Damianakis, A. Bilas, C. Dubnicki, E. W. Felten. Client-Server Computing on SHRIMP. *IEEE Micro*, pages 8–18, Jan./Feb. 1997.
9. Dolphin Interconnect Solutions. *Link Controller LC-2 Specification.* Data Sheet, Dolphin 1997.

10. Dolphin Interconnect Solutions and Sun Microsystems. *Sun Enterprise Cluster Architecture*. Application Note, Dolphin 1998. http://www.dolphinics.com/dolphin2/interconnect/applications/Sun_Cluster_Arc.htm.
11. C. Dubnicki, A. Bilas, Y. Chen, S. N. Damianakis, K. Li. SHRIMP Project Update: Myrinet Communication. *IEEE Micro*, pages 50–51, Jan./Feb. 1998.
12. D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.
13. D. R. Engebretsen, D. M. Kuchta, R. C. Booth, J. D. Crow, W. G. Nation. Parallel Fiber-Optic SCI Links. *IEEE Micro*, pages 20–26, Feb. 1996.
14. R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, pages 12–18, Feb. 1996.
15. R. B. Gillett, R. Kaufmann. Using the Memory Channel Network. *IEEE Micro*, pages 19–25, Jan./Feb. 1997.
16. D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, pages 10–22, Feb. 1992.
17. D. B. Gustavson. The Many Dimensions of Scalability. *Proc. COMPCON Spring'94*, 1994.
18. D. B. Gustavson, Q. Li. The Scalable Coherent Interface (SCI). *IEEE Communications Magazine*, pages 52–63, Aug. 1996.
19. A. Heirich, D. Garcia, M. Knowles, R. Horst. *ServerNet-II: a Reliable Interconnect for Scalable High Performance Cluster Computing.* White Paper. Compaq Computer Corporation, Tandem Division. Sept. 1998. http://www.servernet.com/flat/public/brfs_wps/snetii/snetii.pdf.
20. M. D. Hill, J. R. Larus, D. A. Wood. *The Wisconsin Wind Tunnel Project: An Annotated Bibliography.* Technical Report. Computer Sciences Dept., Univ. of Wisconsin-Madison. Aug. 1999. http://www.cs.wisc.edu/~wwt.
21. R. R. Hoare, H. G. Dietz. A Case for Aggregate Networks. *Proc. IPPS/SPDP'98*. IEEE CS Press 1998.
22. R. W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, pages 1–9, Feb. 1995.
23. R. W. Horst, D. Garcia. ServerNet SAN I/O Architecture. *Proc. Hot Interconnects V*, Aug. 1997.
24. K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill 1993.
25. IEEE Std 1212-1991. *IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses.* The Institute of Electrical and Electronics Engineers, Inc., 1991.
26. IEEE Std 1596-1992. *IEEE Standard for Scalable Coherent Interface (SCI).* The Institute of Electrical and Electronics Engineers, Inc., 1993.
27. IEEE Std 1596.3-1996. *IEEE Standard for Low Voltage Differential Signals (LVDS) for Scalable Coherent Interface (SCI).* The Institute of Electrical and Electronics Engineers, Inc., 1996.
28. IEEE Std 1596.5-1993. *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors.* The Institute of Electrical and Electronics Engineers, Inc., 1993.
29. IEEE Std 1596.7-199X Draft 0.99. *Draft Standard for A High-Speed Memory Interface (SyncLink).* 1999. http://www.SLDRAM.com/FAQ/SyncLinkD0.99.pdf.
30. D. V. James, D. B. Gustavson, B. Fleischer. Serial Express: A High-Performance Workstation Interconnect. *IEEE Micro*, pages 54–65, May–June 1998.

31. J. Laudon, D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Proc. 24th Int'l. Symp. on Computer Architecture*. ACM Press 1997.
32. T. Lovett, R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. *Proc. 23rd Int'l. Symp. on Computer Architecture*. ACM Press 1996.
33. T. D. Lovett, R. M. Clapp, R. J. Safranek. *NUMA-Q: An SCI-based Enterprise Server.* White Paper. Sequent Computer Systems, Inc., 1996. http://www.sequent.com/products/highend_srv/numa_sci.pdf.
34. A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. *Proc. Int'l. Conf. on Parallel Processing.* 1995.
35. *SCIzzL: the Scalable Coherent Interface and Serial Express Users, Developers, and Manufacturers Association.* http://www.SCIzzL.com.
36. S. Scott. The GigaRing Channel. *IEEE Micro*, pages 27–34, Feb. 1996.
37. Siemens AG. *High Performance Computing – HPCLINE.* http://www.siemens.de/computer/hpc/en/hpcline/index.htm.
38. Siemens AG. *RM600 E. Model E30, E70.* Data Sheet. Siemens, Sept. 1998. http://manuals.mchp.siemens.de/servers/rm/rm_us/rm_pdf/rm600e37.pdf
39. Silicon Graphics Computer Systems. *Gigabyte System Network.* Data Sheet. SGI, Nov. 1998. http://www.sgi.com/Products/PDF/2287.pdf.
40. R. Thekkath, A. P. Singh, J. P. Singh, S. John, J. L. Hennessy. An Application-driven Evaluation of the Convex SPP-1200. *Proc. 11th Int'l. Parallel Processing Symposium.* IEEE Computer Society Press 1997.
41. Vitesse Semiconductor Corp. *Compliant Link Controller 1 GByte/sec SCI VSC7201a*. Data Sheet, Vitesse 1996.
42. T. von Eicken, A. Basu, V. Buch, W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *Proc. 15th ACM Symposium on Operating System Principles.* ACM Press 1995.
43. T. M. Warschko, J. M. Blum, W. F. Tichy. ParaStation: Efficient Parallel Computing by Clustering Workstations: Design and Evaluation. *Journal of Systems Architecture.* Vol. 44 (3–4) (Special Issue on Cluster Computing), pages 241–260, Dec. 1997.

2. A Comparison of Three Gigabit Technologies: SCI, Myrinet and SGI/Cray T3D

Christian Kurmann, Thomas Stricker

Laboratory for Computer Systems, Swiss Institute of Technology (ETH),
CH-8092 Zürich, Switzerland
email: {kurmann, tomstr}@inf.ethz.ch
http://www.cs.inf.ethz.ch/CoPs

2.1 Introduction

In 1993 Cray Research shipped its first T3D Massively Parallel Processor (MPP) and set high standards for Gigabit/s SAN (System Area Network) interconnects of microprocessor based MPP systems sustaining 1 Gigabit/s per link in many common applications. Today, in 1999, the communication speed is still at one Gigabit/s, but major advances in technology managed to drastically lower costs and to bring such interconnects to the mainstream market of PCI based commodity personal computers. Two products based on two completely different technologies are readily available: the Scalable Coherent Interface (SCI) implementation by Dolphin Interconnect Solutions and a Myrinet implementation by Myricom Inc. Both networking technologies include cabling for System Area Networking (SAN) and Local Area Networking (LAN) distances and adapter cards that connect to the standard I/O bus of a high end PC. Both technologies can incorporate crossbar switches to extend point to point links into an entire network fabric. Myrinet links are strictly point to point while SCI links can be rings of multiple nodes that are possibly connected to a switch for expansion. In the mean time two Internet technologies emerging from the inter-networking world also arrived at Gigabit speeds—ATM (Asynchronous Transfer Mode) and Gigabit Ethernet. Based on the specification and their history those two alternatives are related to the evaluated technologies Myrinet and SCI.

For a systematic evaluation and comparison of the different Gigabit interconnects a common *architectural denominator* is required. We propose to look for common grounds among the interconnect technologies at three different levels: first for a simple and highly optimized remote load/store operation using all the knowledge about the hardware details (DIRECT DEPOSIT), second for an optimized standard message passing library (MPI/PVM) providing a standard API for specialized parallel programs and third for a connection oriented LAN networking protocol (TCP/IP) catering to the needs of many high level services and middle-ware packages used in distributed computing. We are convinced that a better performance for TCP/IP LAN emulation in

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 39-68, 1999
© Springer-Verlag Berlin Heidelberg 1999

a Gigabit networking environment can make Clusters of PC's (CoPs) a very attractive platform for a large number of PC users. Several high level protocol stacks, middle-ware packages and applications based on TCP/IP sockets are widely available and in most environments it would not be economical to adapt this large and diverse software base to some interconnect specific network interfaces. At a later date the studies can be extended to higher-level services like e.g. NFS and AFS remote file systems, high performance Web servers or SQL servers for distributed databases.

Most previous performance studies remained limited to the measurement and the discussion of maximum transfer bandwidth, minimal ping-pong latency or simply assess the performance of an installed cluster of PCs with a single application. Unfortunately such simplistic studies are still the state of the art in comparing different network technologies. We claim that such studies are inadequate, since most scientific application codes for parallel computers or clusters of workstations have at least some common requirements for computation and communication that could be used for a more general performance characterization. Most parallel and distributed applications of interest deal with large quantities of distributed data in either regular or irregular fashion. While some benefit from a regular layout of their data and store their data in distributed arrays, other applications use fine grain object stores as their distributed data structures. In both cases we encounter a few characteristic communication patterns when arrays or object collections are redistributed or distributed objects are migrated. The regular communication pattern in array transpose primitives is a good generic test case to characterize the strengths and weaknesses of the communication and memory systems. We therefore extend the benchmarks to cover some data types beyond contiguous blocks and incorporate the processing of strided transfers as a more representative memory access pattern for realistic applications.

2.2 Levels of Comparison

The two principal functions of a high performance communication system in parallel and distributed systems are to *move data* and to provide explicit *synchronization* for consistency. This can be done at different abstraction levels with more or less support by the underlying hardware. A common denominator for an evaluation and the comparison of different Gigabit interconnects can be determined by selecting a few common data transfers and by examining the ways those operations can be performed based on different programming models and based on different degrees of support from hardware and software. While at the lowest level the performance results are highly transparent and can easily be related to the specifications of the hardware, the performance figures at the higher levels correspond most closely to what an application can reasonably expect from several installed systems with dif-

ferent hardware, different software and different application programming interfaces. Therefore a comparison at three different levels is proposed:

- **Direct Deposit:** Direct deposit refers to simple, unsynchronized remote load/store operations of varying block sizes and data access patterns. The performance at this level is expected to be closest to the actual hardware performance.
- **Message Passing (MPI/PVM):** MPI or PVM represent the performance of a highly optimized standard message passing library. Carefully coded parallel applications are expected to see the performance measured at this level.
- **Protocol Emulation (TCP/IP):** An emulation of the connection oriented TCP/IP protocol used in the Internet. Users that substitute a Gigabit/s network for a conventional LAN will see a performance comparable to this benchmark.

Direct Deposit with its simple "no fuzz" remote store semantics permits to determine a maximal fraction of the hardware performance that is sustainable by the lowest level driver software for each interconnect technology. For contiguous blocks this figure corresponds to the best published transfer rate. However for non-contiguous blocks the measured rates expose the capability or inability of the hardware to handle fine grained data in communication operations. If the hardware is unable to execute fine grained transfers efficiently, aggregating copies must be used.

At a somewhat higher level the transfer modes of message passing communication with full buffering capability required by a clean implementation of postal semantics are explored. Some common "zero copy" shortcuts typically restrict the semantics of the messaging API to some extent, but speed up the communication in latency and bandwidth. This route is explored as a contrast to the fully functional messaging libraries. MPI or PVM compatible communication libraries are used for the test without checking much for API completeness or standard compliance—which ever one performs best is used. For the final investigation of network performance delivered in the classic, connection oriented protocol setting, a "TCP/IP over LAN" protocol emulation is selected as a test case.

In the second and third "high level" test scenarios the transfer capability for contiguous blocks satisfies the needs of the data representations in the two APIs, but additional copies may occur due to the tricky postal semantics of send and receive calls or due to the requirement for retransmission to achieve reliable connections over a non-reliable interconnect.

2.2.1 Direct Deposit

Conventional message passing programs use the same mechanism (i.e. messages) for control and data transfers. The deposit model allows a clean separation of control and data messages in the implementation [16]. In the deposit

model only the sender actively participates in the data transfer, “dropping” the data directly into the address space of the receiver process, without active participation of the receiver process or any other action causing synchronization. In addition to transferring contiguous blocks the deposit model allows to copy fine grained data directly out of the user data structure at the sender into the user data structure at the receiver, involving complicated access patterns like indexing or strides. The conceptual difference between the Direct Deposit and the traditional NUMA model is that the deposit promotes and assumes aggregation or pipelining of data accesses despite a possibly non-contiguous access pattern that could occur when communication data is placed directly to its final destination in user space.

The deposit transfers can be implemented in software, e.g., on top of an Active Message layer, where a handler is invoked on the receiver to move the data to its final destination. However, our understanding of optimum hardware support for Direct Deposit suggests that a general control transfer in the form of an RPC should be avoided and that a previously asserted synchronization point is sufficient to move the data. Furthermore the functionality of the message handler is fixed and the deposit operation at the receiver only affects the memory system of the receiver.

2.2.2 Message Passing (MPI/PVM)

The MPI and PVM message passing standard are examples of the classical postal model[1]. Both the sender and receiver participate in a message exchange. The sender performs a send operation and the receiver issues a receive operation. These operations can be invoked in either order, blocking or non-blocking. That is, messages can be sent at any time without waiting for the receiver. This enhancement in functionality forces the system to buffer the data until the receiver accepts it. An optimization in several implementations eliminates one of the additional copies in the receiver node by delaying the data transfer until the receive call is invoked. Once that happens, the receive pulls the data from the sender. This optimization can reduce the number of copies but not eliminate them entirely since in the cases that the receiver is not ready the sender must hold the data and copy instead for proper storage management. Generalized postal semantics in message passing always requires buffering in some situations. Only once the programmer agrees to live with certain restrictions on the proper use of the send and receive calls the libraries can eliminate all copies and provide a so called “zero copy” messaging.

Figure 2.1 shows two possible scenarios of restricted and full postal message passing semantics. The left chart shows a case in which the data can be

[1] Unlike PVM the MPI messaging standard does not require a clean implementation of fully buffered postal semantics, but most advanced implementations do provide it in practice.

transferred directly given that certain restrictions on the use of the send and receive calls are accepted. In the scenario the restricted send and receive calls are synchronous and mutually block each other until the transfer completes at both ends. In the case of the right chart some full postal semantics is assumed and also required. Both messages can be sent without looking at the receiver. The messages will be buffered by the communication system until the receiver is eventually ready to accept them.

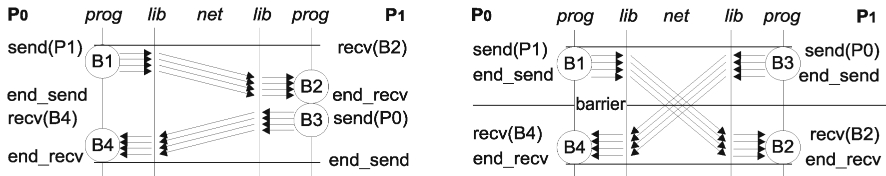


Fig. 2.1. The figure shows two scenarios: first with restricted postal semantics based on synchronous, blocking send calls, which may release the sender only when the receive is executed; second with full postal semantics where non-blocking calls are always permitted, forcing the communication system to buffer the messages until received.

High level MPI and PVM communication functions must be implemented with the lower level primitives that are offered directly by the communication technology. Often the hardware primitives available for an implementation are unbuffered remote loads and stores. For clean higher level abstractions buffering is added in software as an important part of the message passing system. The typical amount of data transferred is usually too large to be stored in special purpose registers of the network interface or in the buffers for data forwarding along the path from a sender to the receiver. Therefore buffering is done in software at a higher level of the message passing library and involves the memory system at the end points. The simplest communication libraries just allow to map the implementation of blocking sends directly to an aggregation of a few remote stores plus a few synchronization primitives to assert that the transfer is valid and complete. Much a simple messaging implementation can be done with little or no buffering storage at the end points. This can be compared with the restricted semantics scenario. The proper execution of non-blocking sends however needs buffering on the receiver side and often leads to an additional copy operation which again largely affects the performance. The study includes a blocking mode of operation for message passing as well as a fully buffered mode of operation as seen in the second message passing scenario. In a serious performance characterization of MPI or PVM implementations both cases must be considered and evaluated separately.

2.2.3 Protocol Emulation (TCP/IP)

Connection oriented LAN network protocols are particularly important for Clusters of PCs and crucial to their commercial viability. Most protocol stacks as well as the ubiquitous socket API are provided by the default operating system of PCs or workstations and many software packages using these protocols and the socket interface are already available. The goal of the CoPs project at ETH is not limited to deploying ever cheaper GigaFlops for applications in computational chemistry, computational biology or computational astronomy, but intends to widen the range of parallelizable applications from scientific codes into databases and Internet servers. Especially for commercial distributed databases or existing object store middle-ware systems it would not be viable to change the standard communication protocols to restricted high speed messaging. For network file systems on clusters of PCs, like e.g. NFS or Sprite, both UDP/IP and TCP/IP services must be provided. With a highly optimized IP communication facility through a Gigabit interconnect a Cluster of PCs can provide high compute performance at an optimal price for a much larger number of programs than a dedicated workstation cluster can do. Traditional clusters of workstations like e.g. an IBM SP/2 offer high communication speeds exclusively to parallel programs that are recoded for message passing communication.

The TCP/IP protocol suite is primarily designed for Internet communication and not particularly well suited for messaging passing communication in parallel systems. However with the underlying IP messaging mechanism it can still offer some fast unreliable, connection-less network services by fragmenting messages into IP datagrams and delivering them according to the IP address scheme. Transport protocols such as UDP and TCP allow to extend communication to different processes of the same end system by a universal port concept called sockets. TCP further enables full duplex communication over a reliable data stream by implementing flow control and retransmission with a sliding windows protocol. The latter functions of TCP are less important in a cluster interconnect (with a proper setup of a high speed interconnect there should be no loss in the switches) but its API is very common if not ubiquitous.

Because the protocol is implemented without specific knowledge of the used hardware, assuming an unreliable network service like Ethernet or Internet respectively, the performance of IP will rarely match the performance of optimized MPI and Direct Deposit protocols. Especially the latency for TCP data transfers is much higher due to connection setup, which might be acceptable or unacceptable to certain applications. The maximum bandwidth of many implementations is lowered by a factor of two or three through the buffering copies as they might be required for retransmission recovery after a possible communication error in a presumably unreliable network.

2.3 Gigabit Network Technologies

For many years, the rapid advances in processor technology and computer architecture have consistently outperformed the improvements of interconnect technology in terms of speed, cost and widespread availability. Over more than one decade Ethernet with 10 MBit/s remained the only main stream networking alternative.

Two of the most promising new networking technologies for interconnecting compute nodes at Gigabit speeds in a high-performance Cluster of PCs are Dolphin's implementation of Scalable Coherent Interface (SCI) [6] and Myricom's Myrinet [3]. Of course this list is by no means complete. There are a few older interconnect technologies that are no longer marketed like e.g. Digital's Memory Channel [7]. There are also other technologies that are newly announced or even shipping, and some we were just unable to evaluate (yet) [17] [9].

Since the mode of operation of an SCI interface connected to a PC is so similar to the hardware of a processor node in the SGI/Cray T3D system we provide a short description of the communication technology of the latter system as a reference for mechanisms, services and performance. The Cray T3D is an old (1992) MPP supercomputing platform, that reached the end of its life cycle in the mean time, but its interconnect is still faster (and still a lot more expensive) than many of the current interconnect solutions in the market of PC clusters and many of the modern, scalable cache coherent NUMA systems.

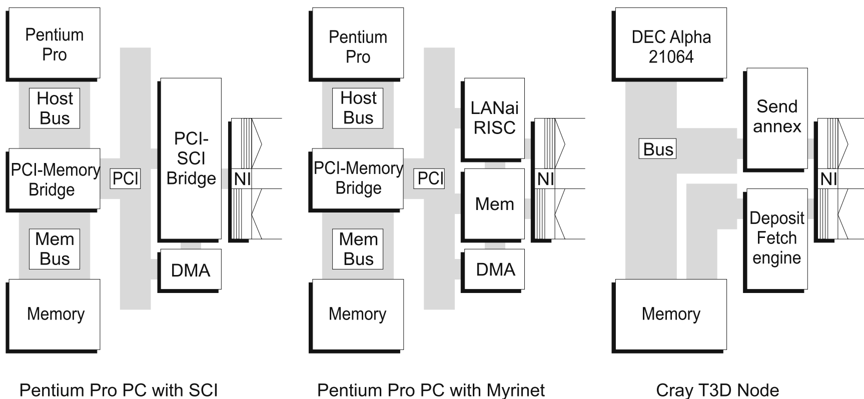


Fig. 2.2. Block diagrams of different network interface architectures. A Pentium Pro PC with either Dolphin SCI Interconnect or Myricom Myrinet PCI adapter and an SGI/Cray T3D node with DEC Alpha Processor and deposit-fetch engine.

A simplified schematic of the network interface circuits used by the Dolphin PCI card, by the Myrinet LANai card and by the T3D parallel processing

node design is drawn in Figure 2.2. The principal difference between the old T3D and the newer commodity interconnects for PCs lies in a tight integration of the network interface with the microprocessor that was still possible for a dedicated MPP machine like the T3D. Standardization issues forced the PC based system to work with an indirect access to the network interface and the remote memory through a widely used I/O bus in the nodes of a parallel system.[2] In the latter designs a "motherboard" chip-set includes a memory controller and an I/O bus bridge. This auxiliary chip-set of a PC assumes the role of a main internal switching hub in the entire system. All I/O operations including those of Gigabit networking must be performed through the auxiliary chip-set and the PCI-bus, which is a data path with lower bandwidth and higher overhead than the primary host and memory bus. I/O buses are geared towards long block transfers of peripheral controllers. They are therefore prone to high startup overheads and are lacking the signals necessary for full shared memory coherency. With the advent of games and virtual reality, the bandwidth requirements for graphics has outgrown the PCI bus and the PC industry reacted with a separate graphics port (AGP) for the newer PCs. Unfortunately networking seems less important to the mass market and has not yet been accommodated with a special port by those motherboard chip-sets. The Cray T3D network uses a direct access to the main memory of the node via a highly optimized deposit/fetch engine that has its own data path to and from the banked DRAM memory system. In addition to that support circuitry a so-called "annex" routes any communication from the caches (built into the microprocessors) directly to the network via a local-to-global address translator, to a few special FIFOs and finally out to the interconnect wires.

2.3.1 The Intel 80686 Hardware Platform

The experimental platform for benchmarking the two commodity Gigabit networks is given by the framework of the CoPs project. The chosen node architecture is based on either a single processor or a twin processor Intel Pentium Pro system running at 200 MHz. The memory system design of this PC platform is based on a two-level hierarchy of caches (L1, L2) that are either integrated on the processor silicon or on a separate die packaged together with the processor. The L1 cache consists of separate 8 kByte for data and instructions. The instruction cache is 4-way set-associative, the data cache is dual ported, non-blocking, 2-way set-associative supporting one load and one store operation per cycle, the cache lines are 32 bytes wide. The L2 cache is a 256 kByte set-associative, non-blocking unified instruction/data cache, which

[2] The first parallel machine that connected its high speed interconnects through a limiting I/O bus, a micro-channel, was the IBM SP/1 in 1992. Its developers correctly anticipated that the immediate compatibility of the interconnect with the latest and fastest processor generation was more important to the success of the SP/1 product than the interconnect speed.

is closely coupled with a dedicated 64-bit full clock-speed backside bus. An Intel 440FX Motherboard chip-set gives the processor access to 64 MByte of SDRAM memory and the 32 bit PCI bus. The processor, the Data Bus Accelerator (DBX) and the PCI Bridge are connected by a proprietary 64-bit 66 MHz host bus (approx. 512 MByte/s). The external PCI bus is 32 bits wide and runs at 33 MHz (132 MByte/s) [10].

2.3.2 Myricom Myrinet Technology

The Myrinet technology is a SAN or LAN networking technology based on networking principles previously used in massive parallel processors (MPPs) [3]. Myrinet networks are built from cables that carry one or two full duplex 1.28 GBit/s channels connecting the host adapter and the switch port point-to-point. Wormhole routing in the switches together with link level flow control on the cable guarantees the proper delivery of every message despite congestion. Retransmissions are not part of the concept and the packet checksums are just for the detection of electrical errors. The 4, 8 or 16 port switches of the Myrinet product line may be connected among each other by one or multiple links permitting configuration in all known interconnect topologies. Packets are of arbitrary length and therefore can encapsulate any type of packet (i.e. Ethernet packets, IP packets, MPI messages) and there is no MTU (maximal transfer unit) for the lowest signaling layer in Myrinet. In the network, link-layer flow control guarantees integrity of the data transfers at the expense of an increased potential of mutual blocking and deadlocks in the switches, which can be addressed by clever routing schemes like e.g. dimension order routing.

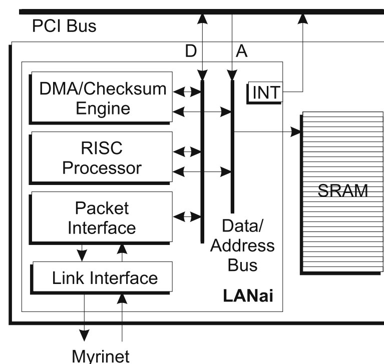


Fig. 2.3. Block diagram of the Myricom Myrinet Adapter.

A Myrinet host adapter (see Figure 2.3) contains a LANai chip with a RISC processor core, several DMAs and the entire network interface integrated in one VLSI chip. In addition to the LANai there are 512 kByte up to

2 MByte of fast SRAM on the adapter card to store a customizable Myrinet Control Program (MCP) and to act as staging memory for buffering packets. Typical MCPs provide routing table management, gather operations, check-summing, send operations, generation of control messages, receive operation, validity checking, scatter operations and the generation of interrupts for the receiving processor upon arrival of a message. Enhanced MCPs can provide performance evaluation tools [5] or gang scheduling support [8]. The RISC processor core is a 32-bit dual-context machine with 24 general purpose registers. One of the two contexts is interruptible by external events which causes the processor to switch to the non-interruptible context. In addition to the RISC processor core the LANai includes three DMA controllers. The LANai can act as a bus master to gather data blocks from or scatter to the host memory over the PCI bus. At the same time, the DMA engine can compute a datagram checksum in its specialized logic unit for CRC (Cyclic Redundancy Checks). The remaining two DMAs are specialized to transfer data between the network FIFO queues and the staging memory on the card. All DMAs can work in parallel which allows pipelined operation at full bandwidth. The DMA can either be initialized by the MCP on the LANai or directly through memory mapped special registers by the main processor.

2.3.3 Dolphin PCI-SCI Technology

Other than Myrinet, the Scalable Coherent Interface (SCI) technology was conceived as an alternative to a processor bus using point-to-point interconnects (see Chapter 1).

We examined SCI adapters for the PCI bus of Dolphin Interconnect Solutions (Revision B and Revision D) [6] (presented in Chapter 3) which are both universal main-stream versions of SCI host adapters without support for full coherency.

2.3.4 The SGI/Cray T3D – A Reference Point

The SGI/Cray T3D node is an interesting example of an old style MPP node architecture specifically designed for a high performance massively parallel processor (MPP). Although the original design is already retired in the age of low-cost PC clusters, it still sets the standards for a simple and fast communication interface. The implementation is done in an expensive, bipolar ECL gate array technology that fits the speed of the full custom CMOS in the microprocessors almost perfectly and no compromises for cost or for standardization were taken. There was no commercial pressure to use a PCI bus between the processor and the network interface in a custom MPP.

The processor board of a Cray T3D comprises a 150 MHz 64-bit DEC Alpha microprocessor (21064), a local memory system, a memory mapped communication port (“the annex”) to send remote stores to the network and

a fetch/deposit engine to execute them at the remote site—see Figure 2.2 in Section 2.3.

The memory of a T3D node is a simple memory system built from DRAM chips without extensive support for interleaving and pipelined accesses. Unlike DEC Alpha workstations, the node has no virtual memory and runs on a slightly modified version of the DEC microprocessor without the functional units for paged virtual memory but with a few segment registers for a low overhead virtual memory mapping of large memory regions instead. The interface between the computation agent and the main memory is centered around an 8 kByte primary cache and a write back queue (WBQ) which are both standard in high performance microprocessors and integrated on-chip. An external read-ahead circuitry (RDAL) can be turned on by the programmer at load-time to improve performance of contiguous load streams from 200 to 320 MByte/s. The local read bandwidth for non-contiguous double word loads is at 55 MByte/s and the latency of a single, isolated load from main memory around 150 ns. For writes, the default configuration of the cache is write-around, and the automatic coalescing for subsequent writes is cleverly derived from the regular operation of the write back queue in the CMOS microprocessor.

The interface between the processor and communication system maps some range of free physical address space to the physical memory of another node in the system; the partner node must be selected as a communication partner with a fixed overhead by modifying the appropriate registers in the annex. The remote stores are a key strength of the T3D design, since once a store operation is issued to the communication port, the communication subsystem takes over the specified address and data, and sends a message out to the receiver. Remote loads are handled in a similar way but they must be pipelined with an external, 16-element FIFO queue for efficiency using the prefetch instructions of the DEC Alpha Microprocessor. The queue requires direct coding support by the programmer or compiler and is therefore rarely used.

At the passive end of a transfer the fetch/deposit engine completes the operation as a remote load/store on behalf of the user at another node. These accesses happen without involvement of the processor at the receiver node (i.e., there is no requirement to generate an interrupt). This circuitry can store incoming data words directly into the user space of the processing element, since both address and data are sent over the network. The on-chip cache of the main processor can be invalidated line by line as the data are stored into local memory, or it can be invalidated entirely when the program reaches a synchronization point. The significant fixed cost for switching the communication partner together with the limitation of only partially coherent caches justifies our classification of the T3D as a highly advanced message passing machine with support for fine grain remote stores.

Transfers from the processor to the communication system can be performed at a rate of approximately 125 MByte/s. In synthetic benchmarks multiple nodes can perform remote stores of contiguous blocks of data into a single node and push these transfers even higher, up to the full network speed (160 MByte/s) [12] without even slowing down the memory accesses of the local microprocessor significantly. In practical systems the number of network nodes (and network interfaces) is only half the number of microprocessors (or processing nodes). If just one of the two processors is communicating at a time, the network can be accessed at up to 125 MByte/s, due to limitations in the processing node, but if both processors are communicating at full speed each processor obtains about 75 MByte/s of bandwidth to access the network. The packetization of remote loads and remote stores in the network is very similar to SCI. Packets are between one and two dozen 16-bit flits.

2.3.5 ATM: QoS – But Still Short of a Gigabit/s

At its beginning the development of the ATM (Asynchronous Transfer Mode) interconnect technology was driven by network computing. Some prominent ATM vendors (e.g. Fore Systems) emerged from high speed network testbeds developed for network computing (e.g. OC-3 at speeds of 155 MBit/s). The general ATM technology with its short packet size and its advanced network control architecture incorporated many new ideas for quality of service guarantees and had excellent interoperability of components between different vendors. Therefore it was suitable for WAN/MAN (Wide and Metropolitan Area Network) networking services and remains the technology of choice when it comes to pure networking applications. However in the mean time ATM lost its competitiveness in the SAN/LAN (System and Local Area Networking) environment by the delayed introduction and excessive cost of Gigabit ATM interconnects. Several years after their announcement the next generation OC-12 (i.e. 622 MBit/s) host-adapter cards for PCs and line-adapter cards for ATM switches are only affordable for network backbones or high end servers – OC-24 (1.2 GBit/s) or even OC-48 (2.4 GBit/s) ATM links would be needed to bring the technology up to a sustained Gigabit/s and to make it competitive with SCI or Myrinet. In fall of 1998, a recent commercial offer for a 16-node fully switched Gigabit/s interconnect priced an OC-12 ATM solution almost a factor of three higher than the corresponding solution based on Gigabit Ethernet.

2.3.6 Gigabit Ethernet – An Outlook

Gigabit Ethernet [14] is the latest speed extension of the ubiquitous Ethernet technology. Its standard war recorded as the IEEE 802.3z in 1998. Ethernet is successful for several reasons. The technology is simple and uncomplicated, and this could potentially translate into high reliability and low maintenance

cost as well as a low cost of entry. For high performance computing Gigabit Ethernet has to overcome several limitations. Its switches are mostly store and forward while the Myrinet switch is wormhole routed. The specified maximum latency is rather increasing than decreasing in advanced switching products and is specified at 20 microseconds for the state-of-the-art switch installed at our site. The 16-port switch backplane is specified to sustain 32 GBit/s switching capacity which will make it nearly impossible to bring the switch to its bandwidth limit. The recently installed host adapter cards of Gigabit Ethernet are PCI based—just like all commodity interconnects discussed in this chapter. During their first year on the market their costs fell already well below the adapters for Myrinet and Dolphin SCI.

2.4 Transfer Modes

2.4.1 Overview

In our description of the possible transfer modes we focus on the performance of moving just data and disregard any difference in amount of local or global cache coherency that the different technologies can offer, since at this point none of the three technologies can offer automatic fully coherent shared memory to support a standardized shared memory programming (like OpenMP) directly in hardware. The direct implementation of a programming model will remain a privilege of either much less scalable or much more expensive systems like bus based SMPs or directory based CC-NUMAs. Also the implication of different network interconnect topologies is a well researched topic and therefore we just assume that a sufficient number of switches is used to provide full bisectional bandwidth for the machines under discussion in this chapter, as this is the case in most smaller systems. It is also clear that all data transfers to remote memory must be pipelined and aggregated into large messages to achieve the full performance given in this report. The pure ping-pong latency of a single word data transfer remains of little interest for a comparison of the sustainable end-to-end throughput experienced by different application programs using different transfer modes and assuming that all processors, co-processors and DMAs are working nicely together.

Until recently the maximum performance of the memory and the I/O system was rarely achieved by the network interconnects. Therefore neither the performance of the I/O bus designs nor the performance of the common system software was optimized enough to work well with Gigabit networking. Those two factors are the principal bottlenecks in today's Clusters of PCs.

A further bottleneck is the lack of local memory system performance in PCs. Memory system performance is not only important to computational efficiency in applications with large datasets, but it is also the key to good performance of inter-node transport of data. While high end MPP node designs

can afford memory systems with special hooks for inter-node communication at Gigabit/s speeds, low end systems must rely entirely on mass market memory systems and standard I/O interfaces (i.e. a PCI bus) for economic reasons.

The main difference between the SCI and the Myrinet network adapters are their default transfer modes and some alternate modes they can operate in. Although the hardware mechanisms involved in transfers between main memory, staging memory and network FIFO queues may be vastly different, the purpose of all data transfers remains the same: to move data from the user space of a sending process to the user space of a receiving process. Most interconnect designs can perform this operation with close to peak speed for large blocks of data and for the special semantics of a zero copy messaging API. However this peak speed is not necessarily a good indicator of the real transfer modes found in common applications. Our model requires that a direct remote memory operation can also include some more complex memory accesses at the sending end and the receiving end, e.g. deal with strided stores. Thus the issues of data transfers can be explored in more depth. A typical application for a complex data transfer operation would be the boundary exchange of an iterative FEM solver working on large, space partitioned, sparse matrices. Figures 2.4 - 2.9 illustrate two transfer options for each of the interconnects discussed earlier. The first transfer mode is mostly processor driven and utilizes the most direct path from memory to the network FIFOs at the sender and from the network FIFOs at the receiver to the memory, while the second mode is DMA driven. The second mode makes a few additional copies on the way to the network interface but uses DMAs to do them most efficiently in parallel to the regular work of the processor nodes involved in communication. The naming for those different mode of transfer is as follows:

Direct Deposit by the processor (direct mapped): The main processor pushes the data directly into the network FIFOs through regular store operations addressed to a special segment of virtual memory, mapped directly to the network interface and through that port on to the memory of the remote processor. Contrary to a common belief, the precise layout or format of the assembly instructions to trigger this remote store or load operation does not matter. It is well conceivable that a parallelizing compiler automatically handles a remote store as two separate stores for address and data, since, unless some magic hardware can deliver full coherency without speeds penalty, the compilers or the programmers will have to know about the local and remote nature of their data references for the sake of better performance optimizations.

This Direct Deposit mode of operation is the native mode for the SCI adapter and for the Cray T3D architectures which both have direct hardware support for remote stores and loads. A Myrinet adapter can only map the control and send registers as well as its local staging memory (SRAM)

into the user address space of the application, but not the entire remote memory segment it communicates to. Direct remote stores are impossible unless the two dedicated MCPs (co-processors) at the sender and the receiver side become involved in moving data. In principle a dedicated control program for those co-processors could shadow the staging memories of the adapters, transfer the data across the network and move the incoming data to the remote memory at the receiving end, so an ensemble of SCI like remote store operations can be emulated for large contiguous or strided blocks of data. This technique does not work too well for one isolated store, but as we will see in a later section, it performs adequately for an aggregation of multiple stores, even with indices or strides involved.

Direct Deposit by the DMA: The application stores its data (and potentially also the addresses) into a reserved, pinned address segment of local memory instead of the mapped remote address space. Starting from there, the DMA of an adapter can pull the data directly into the network FIFO interface or store the received data from the network FIFOs into the host memory respectively.

This mode of operation works very well for *contiguous* blocks of data.

The SCI interfaces in message passing mode can send blocks of the mapped memory using their DMAs. The DMA controller utilizes the most efficient sequence of SCI transactions to achieve highest possible throughput. The Myrinet adapter with its three DMAs on the LANai permits a similar mode of operation. Furthermore there is a bit more flexibility with Myrinet than there is with SCI, since the data can be gathered from small portions, stored at the staging memory and sent directly to the packet interface. The DMAs of Myrinet can be supervised and periodically restarted by the LANai MCP.

Message Passing by the processor: In this mode the main processor of the PC gathers the segmented data words from user space in memory and stores it back to a contiguous buffer of local memory. From this segment of mapped main memory the network card's co-processor or alternatively a network DMA will transfers the contiguous message into the FIFOs and onto the network wires. On the receiver node the message is stored in a contiguous buffer in local memory and unpacked by the main processor.

In this mode the message is processed by the main processor, by a network processor or by a DMA to be finally transmitted into the network FIFO. Measurements indicate that this is the best way of transferring scattered or strided data with Myrinet and this mode therefore can be called the *native* mode of Myrinet.

Message Passing by the DMA: Depending on the available hardware support the buffer-packing/unpacking process can also be done by the DMA. The main processor packs the destination store addresses along with the data into a message. The adapter card fetches the prepared message from memory by a DMA and pushes it into the network FIFO. On the receiver node the

main processor or the network card's message processor reads address and data words and scatters the data via DMA transfers to a segment of main memory. In the same manner the buffer-packing can be done by the DMA of the network adapter provided that the addresses or the pattern of the data words which have to be gathered are simple and known in advance.

The message is processed by a network processor and a DMA at both the sending and the receiving ends. This mode of transferring data can be used if the main processor must stay out of the communication process. Measurements indicate that, depending on specific block sizes and access patterns packing/unpacking by the main processor can be faster than gathering/scattering by the DMA.

2.4.2 "Native" and "Alternate" Transfer Modes in the Three Architectures

To implement the remote memory system performance tests with their complicated strided patterns we can either use the hardware support provided by the communication adapter or the method of packing/unpacking by the main processor. The latter choice always leads to an additional copy operation, but avoids some inefficient single word transfers across the PCI bus. Due to its I/O bus architecture bursts of contiguous data are much faster over the PCI bus.

Myrinet: The Myricom Myrinet adapter with its own RISC processor and its staging memory allows for different scenarios. Figures 2.4 and 2.5 show two schematic flows of data, a direct deposit and a buffer packing/unpacking operation. The LANai processor in the adapter of the receiver can be used to unpack and scatter the data without any invocation of the main processor. Here the DMA between the adapter card and the main memory is the bottleneck when small amounts of data (words) are transferred because the PCI bus arbitration overhead dominates the performance of each transfer. For these access patterns, the packing/unpacking by the main processor and sending the contiguous packed data with a DMA transfer leads to a much higher throughput.

SCI: Figures 2.6 and 2.7 show a schematic flow of data for the Dolphin PCI-SCI Adapters. SCI supports a direct mapped mode which enables transparent access to mapped remote memory segments. As the data is first stored in stream buffers, this mode works perfectly well for contiguous blocks whereas for some unfavorable access patterns, e.g. strided or indexed accesses, only one stream buffer is used, and consequently, the performance drops below 10 percent of the maximum bandwidth. The performance can be increased in the same way as for Myrinet if for strided and indexed accesses the main processor unpacks the data after performing a fast contiguous transfer.

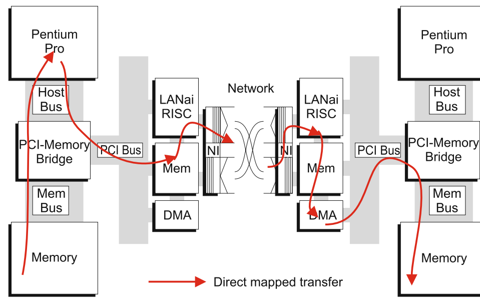


Fig. 2.4. Schematic flow of data on Myrinet. A direct mapped mode for chained transfers can be implemented using the SRAM and the LANai processor on the adapter card. This is the alternate mode for Myrinet.

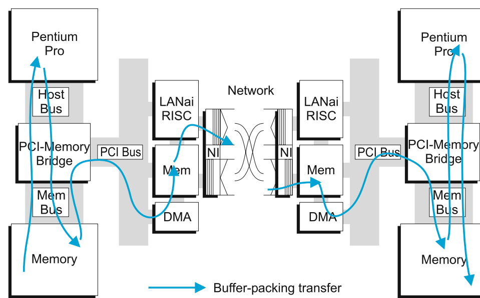


Fig. 2.5. Schematic flow of data on Myrinet with buffer-packing/-unpacking. This is the best message passing mode for any data that must be packed and unpacked by the main processor. It can be labeled as the native mode of operation for Myrinet.

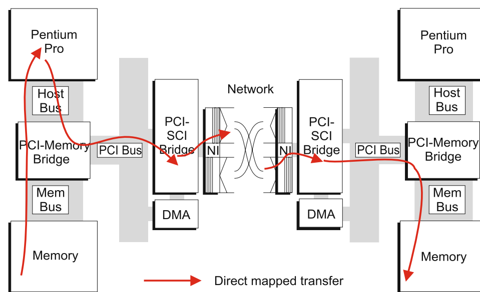


Fig. 2.6. Schematic flow of data with SCI. SCI supports a direct mapped mode which enables transparent access to mapped remote memory segments. This is the native mode of SCI.

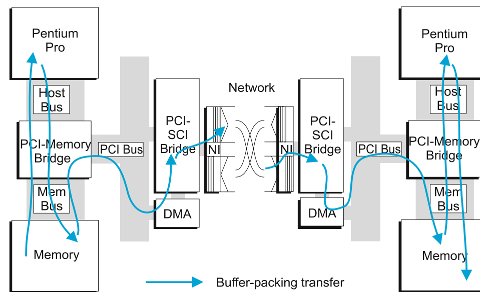


Fig. 2.7. Schematic flow of data with SCI in message passing mode. The main processor packs the data in the local memory and copies contiguous blocks to the mapped remote memory. In addition contiguous transfers can be sped up by a DMA mode which sends contiguous messages to mapped remote memory segments similar to the Myrinet adapter. This is the alternate mode of operation for SCI.

T3D: The T3D offers hardware support to perform direct user-space to user-space transfers for all communication patterns at full speed, contiguous and strided and even irregular indexed (see Figures 2.8 and 2.9). This capability potentially eliminates all buffer packing at the sender and unpacking at the receiver even for the most complex access patterns. The Cray SHMEM library (libsma.a) provides a thin layer to cover the hardware details of Direct Deposit for bare bone contiguous transfers. A buffer packing message passing style interface is provided by the Cray PVM or MPI libraries for a higher level of messaging in system software. While both libraries contain primitives for direct contiguous block transfers, both libraries fail to provide adequate flexibility for transfers of strided and indexed data without prior copies in local memory. However parallelizing compilers like the experimental CMU FX compiler or the Cray production compiler CRAY CRAFT can program the network interface directly to do strided and indexed transfers most efficiently and hereby achieve best performance.

2.5 Performance Evaluation

It is important to keep in mind at which level the benchmark is performed (at the lowest level of deposit or at the highest level of LAN emulation) and which data path (mode of operation) is used to do the transfer. For modes of operation that involve packing/unpacking operations of messages into buffers, the local memory system performance is a very important factor for communication performance. Unlike the simplistic McCalpin loops of the STREAM benchmark [11], the Extended Copy Transfer (ECT) test captures many aspects of a memory hierarchy, including the performance behavior for accesses with temporal and spatial locality by varying working sets and strides or true memory copies with a simultaneous load and store data stream[15].

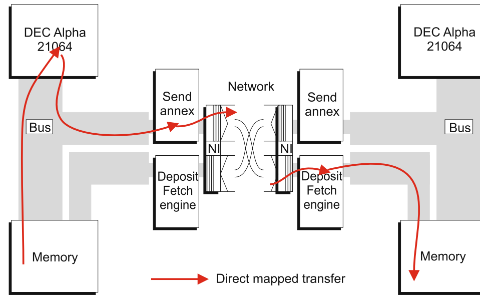


Fig. 2.8. Schematic flow of data on the Cray T3D. Direct transfers can be implemented by mapping the entire remote memory into local address space (through the annex). A special fetch/deposit circuitry handles incoming remote operations without involvement of the processor. This is clearly the native mode of operation for the Cray T3D.

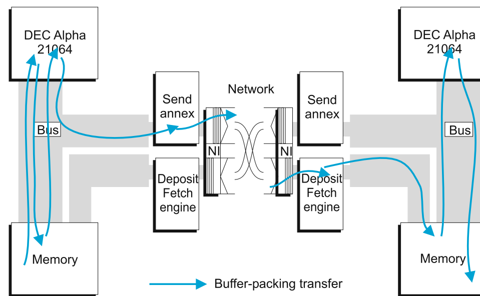


Fig. 2.9. Schematic flow of data on the Cray T3D for the message passing mode. The main processor packs the data in the local memory, copies it as a contiguous block to remote memory and unpacks it again. This is an alternate mode of operation that behaves inferior on all access patterns.

The strides are incorporated based on the fact that end-to-end transfers in compiled parallel programs involve fine grain accesses, such as strided accesses into arrays or a large number of indexed accesses to smaller blocks when gathering/scattering data from/into distributed collections of objects. The ECT method can be used for local and remote memory with full, partial or no support for global cache coherence.

2.5.1 Performance of Local Memory Copy

A good characterization of an interconnect technology is obtained when examining the remote memory system performance figures for different access patterns (strides) and a large working set. The measured copy performance for the same copy operation entirely within the local memory system is included in Figures 2.10 - 2.12 just for an interesting comparison. Since this report is only discussing remote deposit (leaving out remote fetch) all transfers in the performance charts are done by contiguous loads from local memory and strided stores to the same local or remote memory system. The interesting performance numbers for local transfers in Figures 2.10 - 2.12 show that the copy bandwidth of the memory system on the Pentium Pro PCs is only 45 MByte/s and much less than the peak performance of modern interconnects. Copying data in the same memory system with the processor is therefore always a bottleneck and must be avoided at all cost when writing software for fast communication.

2.5.2 Performance of Direct Transfers to Remote Memory

The performance of direct transfers is measured for contiguous blocks (stride 1) and for increasing strides (2-64). The first set of performance curves in Figures 2.10 - 2.12 (filled circle) marks the performance of the most direct transfers by remote store operations to the mapped remote memory or a good emulation thereof. The second curve (hollow circle) marks the performance of highly optimized buffer packing transfers. In this case the transfers were optimized as far as possible. If the DMA was faster, then DMA was used. The relationship between local and remote memory performance can be understood by comparing the performance curves of local memory for the corresponding copy operation to the direct and buffer-packing remote performance curves (triangle).

Deposit on Myrinet: For Myrinet a uniform picture for strided data is observed. The emulation of Direct Deposits for small blocks of data (double words) works with a pipelined transmission of large data blocks with either the LANai or the main processor unpacking the strides (see Figure 2.10). The direct store by the DMA is very much affected by the size of the chunks transferred by one DMA activation. The buffer packing mode with the main processor seems to perform at about memory copy bandwidth whereas the

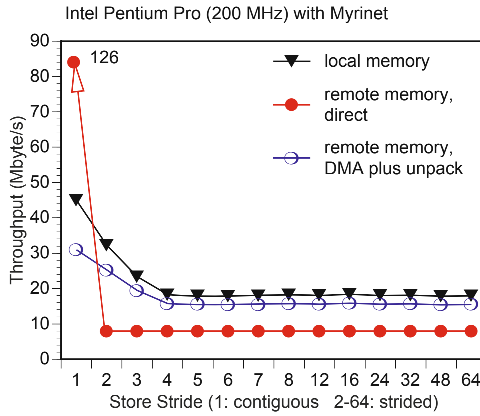


Fig. 2.10. Measured throughput for the Myrinet host adapter using an emulation of direct mapped and buffer packing transfers. As a reference the corresponding performance curve of the local memory system is given (same contiguous loads / same strided stores).

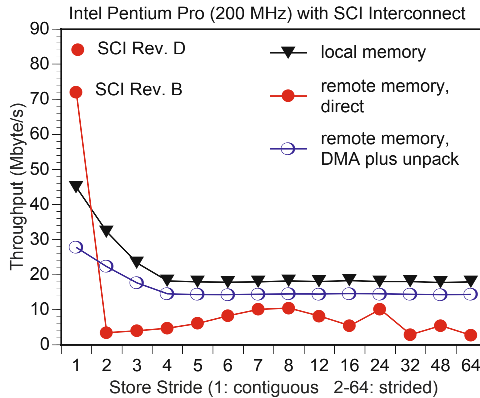


Fig. 2.11. Measured throughput for the Dolphin PCI-SCI host adapter using direct mapped and buffer packing transfers. As a reference the performance of the local memory system is given for the same copy operation (contiguous loads / strided stores).

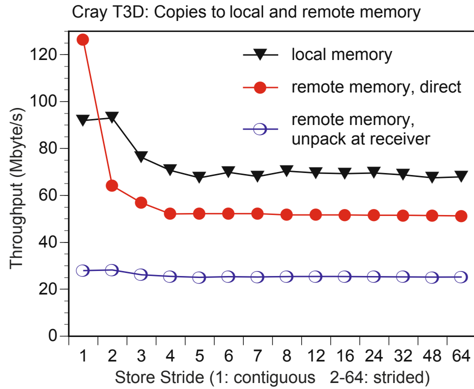


Fig. 2.12. Measured throughput for the Cray T3D using direct mapped and buffer packing transfers. As a reference the performance of the local memory system is given for the same copy operation (contiguous loads / strided stores). Note that two T3D nodes can exchange contiguous data faster than a single node can copy it.

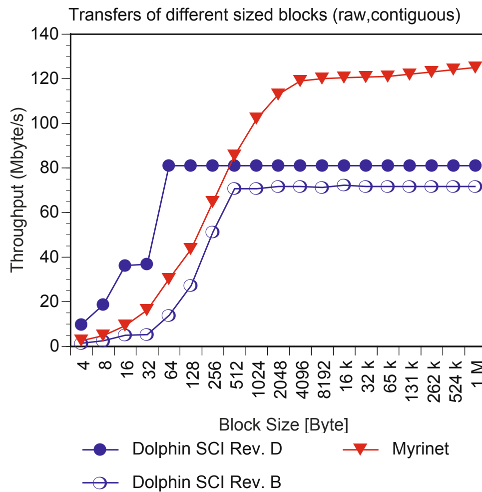


Fig. 2.13. Fastest transfers of different block sizes for Myrinet and SCI.

direct DMA transfers suffer from the overhead of too many DMA initializations and too many PCI bus arbitrations. The buffer packing, native mode can fully use the DMAs to boost the case of large contiguous blocks.

DMA transfers are very much affected by the block size. The performance for different block sizes for Myrinet and SCI is compared in Figure 2.13. It turns out that both adapters have the same problems with small blocks but it should be noted that the second generation of “CluStar” SCI adapters performed much better than the first release when it comes to startup overhead for small transfers.

Deposit on SCI: With SCI interconnects reasonably good performance for contiguous blocks can be observed in direct transfer mode (see Figure 2.11). This is when the eight stream buffers work optimally. For strided data the performance of remote stores on PCI collapses to well under 10 MByte/s and appears to be unstable. The sloped curve from stride 2 to stride 8 can be explained by the mechanics of the stream buffers. For transfers with stride 2 only one stream buffer appears to be used. The direct mapping of the subsequent even word-aligned remote memory addresses to the same 64-byte stream buffer must lead to a sequence of non-pipelined single word transfers. The buffer is always sent directly with only 8 bytes of data and the next value addressed to the same buffer has to wait until a returning acknowledge releases the buffer again. Despite its apparent support for direct transfers of single words in SCI, it turns out to be more efficient to pack and unpack a communication buffer than to execute strided transfers directly. The resulting transfer mode - performance tradeoff is similar to Myrinet.

Deposit on the Cray T3D: The Cray T3D offers (or more precisely offered at its time) a remarkably different performance picture (see Figure 2.12). It turns out that for a Cray T3D it is always best to execute a data transfer in direct mode. Buffer packing includes copies and those never accelerate any transfers regardless of their regularity. For contiguous blocks a direct copy to remote memory is even faster than a local copy from and to memory. This is not surprising since two memory systems, one at the sender side and one at the receiver side, are involved in a single data transfer in the remote case. But even without the advantage of using two memory systems for a copy transfer, the remote copy remains strictly faster than the local one as proven in some unpublished experiments that featured simultaneous send and receive actions on the same node.

2.5.3 Performance of MPI/PVM Transfers

The performance of the higher level transfers indicates how well system programmers can work with the hardware. A full function standard message passing library with buffering for true postal message passing and a reduced zero copy library is used to determine the performance of MPI/PVM transfers for reasons explained in Section 2.2.2. The following evaluations use

two different tests exposing (a) the performance for libraries with full postal functionality including buffering and (b) the performance for libraries with reduced functionality based on zero-copy transfers.

MPI on Myrinet: On Myrinet the BIP-MPI package [13] is used to benchmark message passing performance. BIP (Basic Interface for Parallelism) is a high performance library with a possibly simplified data transfer semantics. The code was developed at the ENS (Ecole Normale Superieure) at Lyon, France. Although this implementation might eventually fall somewhat short in terms of completeness and compliance with the full MPI standard, it seems to provide a stable API for all important functions of basic message passing. BIP-MPI is a modified MPICH version using the lower messaging layers of BIP to drive the Myrinet network hardware. The performance of BIP-MPI (see Figure 2.14) matches the raw performance of 126 MByte/s for blocking sends and receives. This performance figure is measured with large blocks (> 1 MByte). Half of the peak performance can be reached with messages of roughly 8 kByte in size. The performance results are summarized in Figure 2.17.

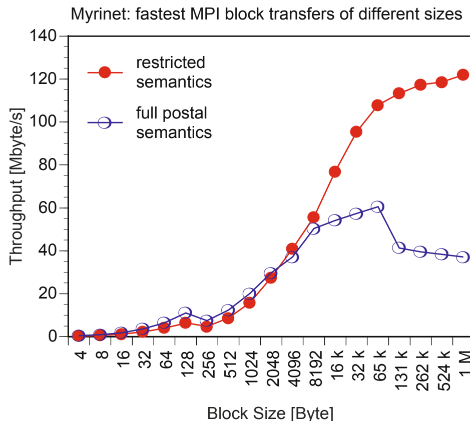


Fig. 2.14. Measured throughput of BIP-MPI transfers over Myrinet for restricted and full postal semantics.

To determine the performance of message passing with the non-blocking send and receive calls, the sends are posted before the corresponding receives. In this case MPICH enforces some buffering and the performance drops to about the local memory copy. An optimized data path for small transfers below 64 kByte uses the LANai staging memory for buffering, saves the additional local memory copy and results in notably better performance. If two sends are posted before the receive, the peak performance for buffered transfers is measured at 32 kByte blocks, since two blocks have to be buffered in the buffer pool on the adapter card.

MPI on SCI: For the message passing tests with SCI a fully standardized MPI version for the Solaris Operating System named ScaMPI is provided by Scali Inc. [1]. The graphs in Figure 2.15 sketch a performance picture for SCI that is quite similar to the one of Myrinet. The peak performance again matches the raw performance at about 72 MByte/s for blocking sends and receives measured with large blocks (> 1 MByte). Half of the peak performance can be reached with messages of roughly 2 kByte size which fits to the fact that SCI is capable of transferring small packets at full performance. For the non-blocking calls the performance again drops to the local memory copy bandwidth.

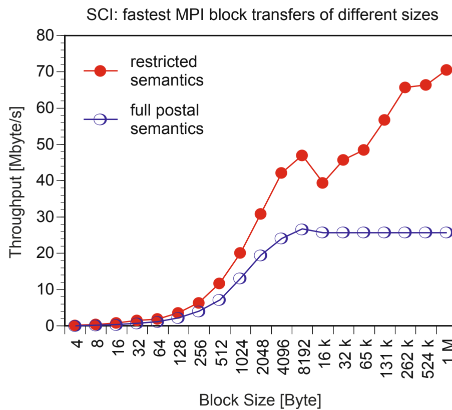


Fig. 2.15. Measured throughput of MPI transfers over an SCI interconnect for restricted and full postal semantics.

MPI and PVM on the SGI/Cray T3D: For the Cray T3D the performance measurements are carried out with PVM 3.0 instead of with the MPI library. PVM 3.0 is a highly optimized, fully functional message passing library supplied by the vendor upon delivery of their first T3D system in Pittsburgh in the fall of 1993. A maximum of 30 MByte/s can be sustained for large blocks (> 1 MByte) and half of the peak bandwidth is achieved with messages as small as 2 kByte size.

Our team has no longer worked on the Cray T3Ds after the first Cray T3E MPP was installed at the Pittsburgh Supercomputing Center in 1996. Therefore our measured PVM results are complemented with the better results of a later MPI implementation for the Cray T3D coded by the Edinburgh Parallel Computing Center in cooperation with SGI/Cray Research (CRI) [4] (see Figure 2.16). The EPCC MPI implements the full MPI specification and was developed using the Cray SHMEM primitives for the T3D. The measured performance corresponds to about the performance of BIP-MPI over Myrinet with blocking calls. Blocking calls do not need any buffering whereas measurements with non-blocking calls are slowed down by an additional copy in

the local memory system. For some reason the blocking calls have a slightly higher overhead. We cannot explain this because we collected published data measured by other groups to draw this comparison.

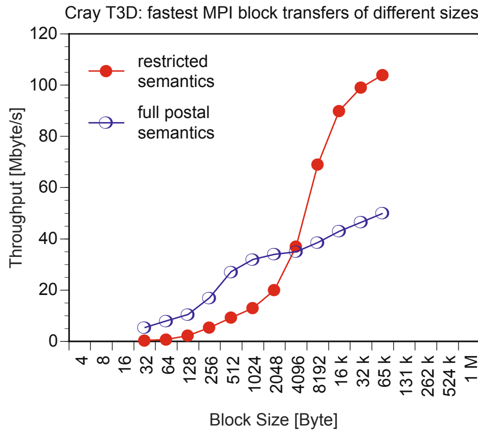


Fig. 2.16. Measured throughput of EPCC-MPI transfers for the T3D for restricted and full postal semantics.

2.5.4 Performance of TCP/IP Transfers

For a more significant growth in market share for high end Clusters of PCs, it is crucial to port traditional applications quickly and easily to the new platform. Moving portable codes to a better platform is also important when performance critical applications are moved from a Beowulf [2] class system to a high-end cluster with a better interconnect. Such an upgrade in communication performance can be done by substituting a conventional LAN network (e.g. switched 100BaseT) by a Gigabit interconnect, especially in the booming area of Internet content servers and for the important market of distributed databases. Keeping the services and APIs stable and compatible during upgrades is guaranteed by protocols like TCP/IP. In the TCP/IP mode of operation the performance of a fully standardized protocol stack is essential. Therefore the most standard and best IP emulation packages available for each interconnect technology are examined (with the small exception of the T3D, where applications using IP did not make much sense at its time).

IP on Myrinet: Myricom offers a fully compliant TCP/IP protocol stack that transfers data at 20 MByte/s. BIP-TCP [13] improved this performance by using the "zero copy" BIP interface so that about 40 MByte/s are reached. The BIP implementors use the original Linux protocol stack and substituted

the transfer mechanism using their BIP message passing system in the lower layers.

IP on SCI: For the SCI Revision B cards under test we could only reproduce 13.2 MByte/s with a non-optimized NDIS driver provided by Dolphin Interconnect running under Microsoft Windows NT. We also tested a TCP/IP emulation written by researchers at the PC^2 at the University of Paderborn. The code is quite similar to the BIP-TCP and also runs under Linux. Instead of the BIP layer, mechanisms using remote mapped segments were used to implement the transfer layer in the standard Linux TCP/IP protocol stack. Dolphin cards (Revision D) transfer up to 22 MByte/s over that TCP stack on the Pentium Pro platforms and about 30 MByte/s on a Pentium II platform with BX chip-set.

IP on the SGI/Cray T3D: A T3D is unusually connected to a Cray C90 or J90 vector processor as attached massively parallel multiprocessor to complement the vector processing supercomputing capability. The T3D processing nodes execute only threads in SPMD style tightly controlled by the host. The small runtime system of the hosts did neither support an IP number nor a socket API. A TCP/IP environment would not make much sense in that context.

IP on Gigabit Ethernet: Gigabit Ethernet is backward compatible with 100 MBit/s Ethernet. Our first installation came with existing protocol stacks for TCP/IP through the use of a standard device driver interface. Those implementations delivered out of the box around 40 MByte/s per link of TCP/IP performance measurable with the “Netperf” utility.

2.5.5 Discussion and Comparison

A summary of the measured throughput for the three technologies at different levels is given in Figure 2.17. The Direct Deposit performance is compared to the MPI bandwidth with both blocking and non-blocking message passing calls. The performance results confirm that the blocking MPI bandwidth roughly matches the raw performance of the Direct Deposit whereas the non-blocking calls with buffering semantics force Myrinet, SCI and the T3D into copying. The performance drops down to the copy performance of the local memory system in the corresponding Pentium Pro or DEC Alpha node architectures. The TCP/IP performance on both PCI based technologies (Myrinet and SCI) reflects the same results and again confirms the overhead of copies in the main memory.

The efficiency of the network interface logic can be evaluated by the ratio of raw data speed on the wires over the maximum throughput sustainable between a sender and a receiver (see Figure 2.18). Assuming highly optimized drivers for this test the best measured or published values were used for this comparison. The raw hardware speed is specified as 1.6 GBit/s for

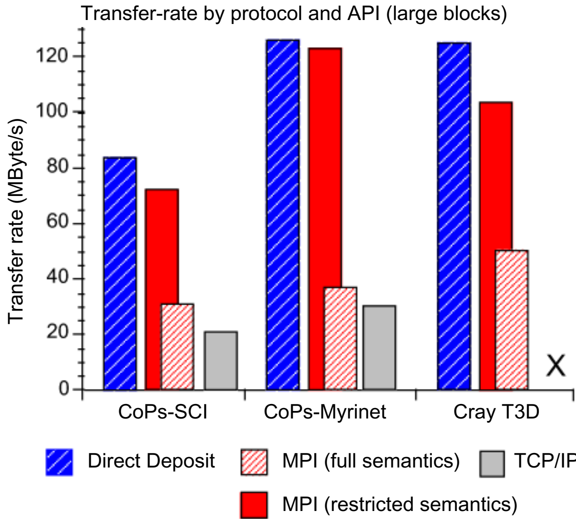


Fig. 2.17. Measured performance of different protocols and APIs over SCI and Myrinet with reference to the Cray T3D. Direct Deposit performance is compared to MPI/PVM performance. For the T3D the performance of the PVM 3.0 is listed as the best library supplied by the vendor. TCP/IP implementations are provided for both PCI adapter cards but not for the T3D.

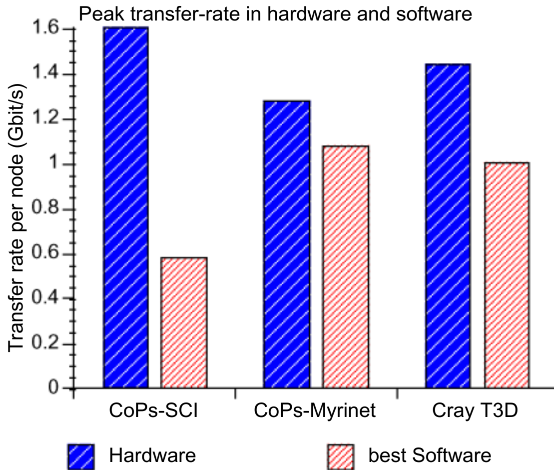


Fig. 2.18. Comparison of communication efficiency in the three evaluated technologies. Conclusions about the efficiency are drawn from a comparison of the raw hardware speed vs. the best possible throughput in software (low level primitives).

SCI and 1.28 GBit/s for Myrinet. For the SGI/Cray T3D the channel width of its interconnect is known to be 16 bits plus handshake lines but to our best knowledge the precise clock on the interconnect remained unpublished. In some experiments the data can be pushed at a rate of approximately 160 MByte/s over a single link. The best packet layout is known to be 4-8 header flits and 16 flits of payload—therefore its raw link capacity must lie between 1.28 GBit/s and 1.92 GBit/s. In all cases full duplex operation at this speed must be assumed and is required for a proper characterization. For most data transfers in real applications it takes the work of a sender and receiver to make the entire transfer happen.

2.6 Summary

The performance of direct-deposit remote memory operations indicates excellent performance on both low-end Gigabit/s technologies for a few simple cases. The transfer rates typically peak near the bandwidth limits of the PCI bus or the interconnect speed and are in fact comparable to the rates seen in traditional MPP supercomputers. Considering the drop in price for a compute node in a parallel system during the past five years this is a remarkable achievement of commodity interconnects. However such good performance is only achieved for the most simple transfer modes and in straightforward communication scenarios like direct deposits of contiguous blocks of data by remote stores or also in libraries with restricted MPI semantics. For strided remote stores or for remote loads of single words, the performance figures of the SCI and Myrinet interconnect collapse, while the traditional MPP can do even those cases at acceptable copy bandwidth.

In PCI bus network adapters the DMA transfers suffer from the overhead of too many DMA initializations and too many PCI bus arbitrations. This problem can be bypassed in a buffer-packing mode with a main processor performing gather and scatter operations at about memory copy bandwidth. For the implementation of message passing libraries with buffering semantics (e.g. MPI) the performance of the Myrinet interconnect is reduced to the local memory system bandwidth while the traditional MPP has a better memory system and can do those cases at better speeds. Similar limitations due to copies in the local memory system occur in the IP-over-LAN emulation transfers in most cases. Because of an excellent implementation of BIP/IP, the Myrinet has a slight advantage over SCI in TCP/IP performance and delivers 40 MByte/s on TCP streams. This number was also matched by an experimental Gigabit Ethernet installation with a 16-port Cabletron Routing Switch and Packet Engines host adapter cards. Those results mean an immediate increase of performance by a factor of 4-8 when upgrading from a switched 100BaseT network to a Gigabit interconnect.

References

1. Scali AS. *ScaMPI User's guide*, 1997. http://www.scali.com/html/scampi.html.
2. D. J. Becker, D. Sterling, T. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proc. ICPP Workshop on Challenges for Parallel Processing*, Oconomowc, Wisconsin, U.S.A., August 1995. CRC Press.
3. N. J. Boden, R. E. Felderman, A. E. Kulawik, Ch. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet - A Gigabit per Second Local Area Network. In *IEEE Micro*, volume 15(1), pages 29–36, February 1995.
4. K. Cameron, L. J. Clarke, and A. G. Smith. CRI/EPCC MPI for T3D. In *Proc. 1st European Cray T3D Workshop*, Sept 1995. http://www.epcc.ed.ac.uk/t3dmpi/Product/Performance/index.html.
5. Y. Chen, A. Bilas, St. N. Damianakis, C. Dubnicki, and K. Li. UTLB: A Mechanism for Address Translation on Network Interfaces. In *Proc. 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 193–204, San Jose, October 1998. ACM.
6. Dolphin Interconnect Solutions. *PCI SCI Cluster Adapter Specification*, 1996.
7. M. Fillo, and R. B. Gillet. Architecture and Implementation of Memory Channel 2. *Digital Technical Journal*, 9(1), 1997.
8. A. Hori. Highly Efficient Gang Scheduling Implementation. In *Proc. Supercomputing'98*, Real World Computing Partnership, Nov 1998. ACM/IEEE.
9. GigaNet Inc. http://www.giganet.com.
10. INTEL Corporation. *INTEL 440 FX PCISET*, 1996.
11. J. D. McCalpin. Sustainable Memory Bandwidth in Current High Performance Computers. Technical report, University of Delaware, 1995.
12. R. Numrich, P. Springer, and J. Peterson. Measurement of Communication Rates on the Cray T3D Interprocessor Network. In *Proc. HPCN Europe '94, Vol. II*, pages 150–157, Munich, April 1994. Springer Verlag. Lecture Notes in Computer Science, Vol. 797.
13. L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. Technical report, LHPC and INRIA ReMaP, ENS-Lyon, 1997. http://lhpca.univ-lyon1.fr/.
14. R. Seifert. *Gigabit Ethernet : Technology and Applications for High-Speed LANs*. Addison-Wesley, May 1998. ISBN: 0201185539.
15. T. Stricker and T. Gross. Global Address Space, Non-Uniform Bandwidth: A Memory System Performance Characterization of Parallel Systems. In *Proc. 3rd ACM Conf. on High Performance Computer Architecture (HPCA)*, 1997.
16. T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers. In *Proc. Intl. Conf. on Supercomputing*, pages 1–10, Barcelona, July 1995. ACM.
17. W.-D. Weber, St. Gold, P. Helland, T. Shimizu, Th. Wicki, and W. Wilcke. The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers. In *Proc. 24nd Intl. Symp. on Computer Architecture (ISCA)*, pages 98–107, August 1997. Product: http://www.fjst.com/.

Part II

SCI Hardware

Due to the complexity of the SCI interconnect standard and the intricacies of building a hardware distributed shared memory (DSM) on a cluster of off-the-shelf workstations or PCs, SCI hardware implementations are far from trivial, even if they need not cover the SCI cache coherence protocols. The SCI hardware relevant for this book are SCI adapter cards plugging into a standard workstation or PC I/O bus, like Sun's SBus or PCI. Using these SCI adapters, high-speed point-to-point links and, possibly, SCI switches, off-the-shelf nodes can be connected into a high-performance compute cluster.

Implementations of such SCI adapter cards need to address a number of issues, the most important of which are: implementation of the physical-layer and logical-layer SCI protocols; fault management; conversion from the I/O bus protocol to the SCI protocol and vice versa; realization of the 64-bit SCI address space (hardware DSM) and translation of the 32-bit physical node addresses into SCI addresses and vice versa; mechanisms to import, export, and mutually map memory segments of the nodes to instantiate the DSM; providing transparent accesses for the CPUs to data living in the SCI DSM; avoiding cache coherence problems on modifications to such data; delivering high communication performance to the software.

Only a few implementations of SCI adapter cards with these capabilities have emerged so far. Two of them are described in this part of the book, one commercial adapter, the other developed in a research institution.

Chapter 3 describes the commercial SCI cards, developed by Dolphin Interconnect Solutions, Oslo, Norway, and Framingham, MA, USA. Dolphin's adapters have become wide-spread: companies like Sun Microsystems, Scali A.S., and Siemens AG incorporate Dolphin technology into their high-performance cluster systems, and almost all of the research and development projects described in this book are based on these cards. The chapter covers both SBus and PCI adapters and describes the solutions Dolphin has developed for the issues mentioned above. For example, the physical layer protocols are processed by a single chip, the so-called Link Controller, now in

H. Hellwagner, . Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 69-70, 1999
© Springer-Verlag Berlin Heidelberg 1999

its second-generation implementation (LC-2). Furthermore, innovative concepts to boost performance, like write combining and the notion of streams, have been introduced by these adapters.

Another PCI-SCI card, described in Chapter 4, has been developed by researchers at Technische Universität München (TUM). The main motivation behind this prototype SCI adapter card has been to develop a piece of hardware that can readily be extended by a hardware monitor at a later stage. Within the framework of the SMiLE project at TUM, this monitor is to be used as a basis for performance analysis and debugging tools. The basic design and implementation of the adapter are described in this chapter, whereas the monitoring concepts are introduced in Chapter 24.

It is worth mentioning that there is yet another prototype implementation of a PCI-SCI adapter card, developed some years ago by CERN and associated institutions within the high energy physics research and demonstration project RD24. Further information can be found at
`http://nicewww.cern.ch/~hmuller/sci.htm`.

3. Dolphin SCI Adapter Cards

Marius Christian Liaaen, Hugo Kohmann

Dolphin Interconnect Solutions, Oslo
email: {info,support}@dolphinICS.no
http://www.dolphinICS.no/

3.1 Introduction

The Dolphin Interconnect Solutions SCI adapter cards have been available since 1993 and have been in use in many applications at many different sites. The product line started with a quite simple SBus card with limited performance and feature set and continued with more sophisticated cards for the PCI bus. All cards are based on the Dolphin Link Controllers LC-1 and LC-2 (LC) [1] which are also used in many other applications like the Dolphin SCI switches and other SCI hardware described in this book.

The SBus cards were developed in cooperation with Sun Microsystems to be used in Sun's clustering solutions. The development of the PCI-SCI bridge was initiated by Dolphin to enable connecting SCI to a more widely used bus; a cooperation with Siemens Nixdorf was established to achieve this after the selection of SCI as the I/O interconnect for Siemens midrange servers. The cards are available for 32-bit PCI at 33 MHz (PCI32) and 64-bit PCI at 33 MHz (PCI64). The PCI64 is a superset of PCI32. The description of PCI32 also applies to PCI64 except when explicitly stated otherwise. A 66-MHz, 64-bit version of the PCI card will be available in the near future.

This chapter describes how the SCI adapters work and what kind of applications the cards can be used in. Issues about using shared memory over the I/O bus are also discussed. An overview of the different software components and interfaces of Dolphin's portable SCI driver will also be given in this chapter.

3.2 Overview of the Adapter Cards

The Dolphin adapter cards allow direct mapping of memory accesses from the I/O bus of a machine to the I/O bus and into the memory of a target machine. This means that the memory in a remote node can be directly accessed by the CPU using store/load operations giving the possibility to bypass time-consuming driver calls in the applications. This is called shared memory or Remote Memory Access (RMA). The high latency of accessing the remote memory as compared to local memory, does not make it very attractive to share program variables over SCI, since maintaining cache coherence is not

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 71-87, 1999
© Springer-Verlag Berlin Heidelberg 1999

possible on the I/O bus. A remote CPU read will stall the CPU, but writes are posted such that the latency is minimized on writes. Sufficient data buffers are available on the adapters to hide the latency from the local CPU. Message passing using a write-only model fits very well into this scheme, offering a low-latency, high-bandwidth and reliable channel that makes it possible to implement efficient message passing interfaces like MPI [9], PVM [8], or VI Architecture [10], as discussed in Section 3.11. The cards have been designed to perform well on SCI writes supporting this type of interfaces.

The Direct Memory Access (DMA) engine is capable of moving memory from a local memory buffer to a remote memory buffer, not using CPU cycles for the transfer.

Dolphin SCI cards only support non-coherent SCI transactions and are basically using SCI as a high bandwidth, low latency and reliable channel.

Connecting I/O buses together using SCI will logically form a long I/O bus where all devices (CPUs, disk controllers, etc.) can reach all other devices and memories. The local I/O space (32 bits) is mapped to a 64-bit SCI address space where each address points to an SCI card elsewhere in the network (see Figure 3.1). Exposing all resources on all I/O buses to each other is of course a security risk which is minimized by hardware mechanisms and cluster software.

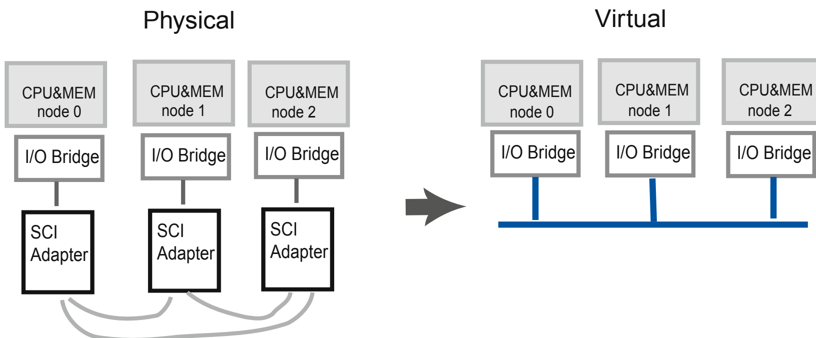


Fig. 3.1. Connecting I/O buses using SCI

The cards are divided into two parts: the Link Controller (LC) which transports the packets over SCI, and the Transaction Controller (TC) which takes care of packet building (data transfer), address mapping, DMA, interrupts, error handling, etc. This chapter will describe the Transaction Controller in more detail.

The LC has a back-end interface to SCI called B-Link, which is a packet based, multi-agent, split-transaction bus, 64 bits wide and with a simple control. B-Link is basically SCI presented in a different format (where the

SCI packets are encapsulated) to utilize a multi-drop bus. Echo packets and synch packets (link layer) are never seen on this bus.

The TC on the SCI cards implements a bus bridge between SBus/PCI and B-Link. Such a bridge can be implemented in many ways from the very simple to a very sophisticated high performance device. The first TC for the SBus was based on two Xilinx FPGAs and multi-port SRAMs but implements DMA and RMA with good performance. The next generation PCI controller was implemented in an ASIC giving considerable room for enhancements. A basic block diagram of the PCI card is shown in Figure 3.2. The PCI-SCI-Bridge ASIC (PSB32/64 [2]) was designed to accommodate both cluster applications and remote I/O. This has given the PSB ASICs some capabilities that make the PCI-based SCI cards also useful in nodes without CPUs.

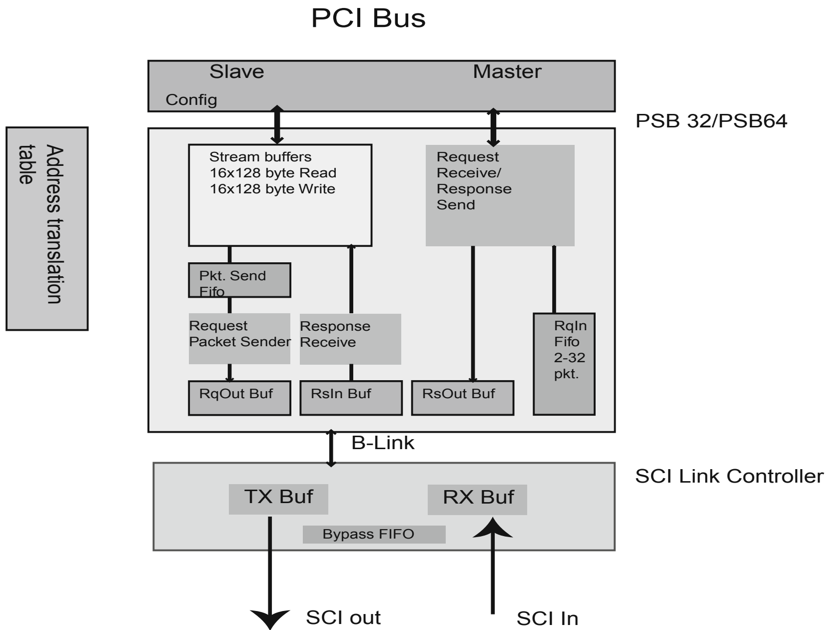


Fig. 3.2. Basic block diagram of the PCI-SCI card

3.3 Operating Modes of the SCI Cards

Slave Mode. When the SCI card operates as a slave on the I/O bus, it will translate I/O bus transactions into SCI packets. The nature of the I/O bus and SCI (bus) is different. SCI is a split-transaction, point-to-point interconnect and the I/O bus uses only single one-phase transactions. The card must handle these differences in an efficient way.

The transfer sizes on SBus and PCI do not match those of SCI which is restricted to 1-byte through 16-byte or 64-byte transfers. (256-byte transfers are not implemented in current cards.) SBus has transfer sizes that easily map into this scheme. PCI is not that simple to work with since there are no fixed transfer sizes. This makes it difficult to build a bridge that works as efficiently as possible under all conditions.

The logic that transfers data into and out of the buffers onto the I/O bus must do this in an efficient way to make the transfer as fast as possible. We have used techniques like write gathering, speculative read buffering and packet prefetching to make I/O transfers work efficiently on SCI. Especially on PCI this has been challenging due to the large number of burst types and the demand to make devices even with small FIFOs or host bridges with limited write-combining capabilities achieve good performance over SCI. Modern PCI devices and host bridge designs are typically capable of very long bursts.

Master Mode. Transforming SCI transactions to the I/O bus is a simpler task than the other way round. The SCI transfer sizes of 1–16 and 64 bytes translate easily to I/O bus transactions on the I/O buses.

DMA Mode. The DMA engine utilizes both the slave and the master mode of the SCI card. It requests transfers both from the I/O bus and SCI.

3.4 SCI Requester

When operating as a slave, the SCI card represents an SCI requester. Its operation involves performing address mapping, generating SCI packets and keeping track of transaction progress.

3.4.1 Address Mapping

Address mapping is the major issue when bridging from an I/O bus like SBus or PCI to SCI. An I/O bus typically has a 32-bit address space. The mismatch compared to the 64-bit address space of SCI must be coped with using address mapping. The address mapping must be flexible and efficient to handle many concurrent mappings. The mapping is based on pages with an Address Translation Table (ATT) containing a map entry (page descriptor) for each page describing the actual mapping of the page.

The page descriptor defines the full SCI address (16-bit node ID and 48-bit address offset). In addition it may also define transfer attributes like write gathering on PCI, use of *dmove* SCI commands (for efficient unreliable transfer), write before read ordering, generation of SCI lock operations, etc. Figure 3.3 shows the basic address mapping from the I/O bus to SCI.

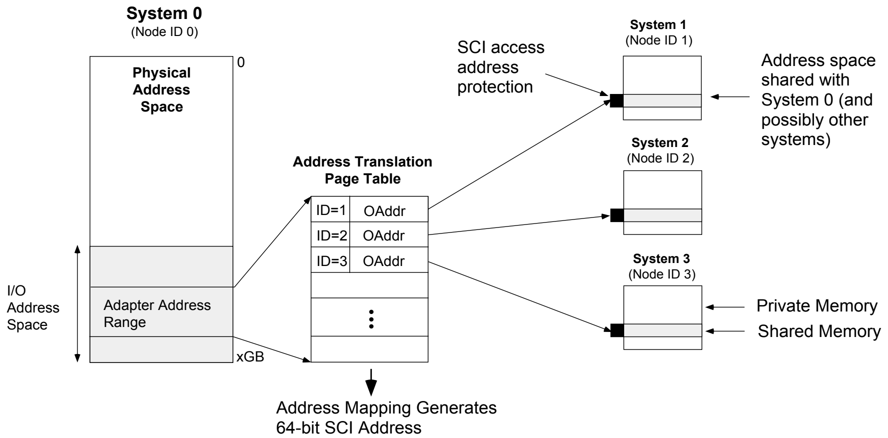


Fig. 3.3. I/O to SCI address mapping

Address mapping on the SBus card. The SBus card has a mapping table with 1025 entries. It uses 64-bit ATT entries and has the capability to map to the whole 64-bit SCI address space. The map page size is 256 kByte.

Address mapping on the PCI32 card. The PCI32 card has the mapping table in SRAM, but it keeps an ATT cache of the last used ATT entry for each stream. The PSB chip replaces the entry automatically on demand.

The page size is 512 kByte and the number of usable entries in the ATT depends on the memory requirements of the board (from 16 MByte up to 2 GByte can be configured, depending on the application).

Address mapping on the PCI64 card. The mechanism works basically the same way as on PCI32, but the page size ranges from 4 kByte to 512 kByte, depending on the memory demand (32 MByte to 2 GByte). There are always 4096 entries available for use. The card supports PCI Dual Address Cycles (DAC) (bypassing the ATT lookup mechanism) giving direct access to the 64-bit SCI address space (limited to 42-bit address offset).

3.4.2 SCI Transaction Handling

It is necessary to have control over the requests the cards are sending to SCI, in order to ensure that no packets are lost and no data errors or other errors go undetected. There is no room for failure in this respect regardless of which approach is used. The SBus and PCI cards are using different methods.

Counting packets on the SBus card. The transaction ID field in the SCI packet together with the source ID uniquely identifies a packet on SCI. This fact makes it useful to keep track of the number of outstanding packets by tagging each packet with a unique transaction ID. By discarding all other

information than that a packet was sent, simple hardware requiring little buffer capacity can be implemented.

Keeping full context on the PCI card. This is a much more powerful approach that makes it possible to track all outstanding packets down to destination node, address offset, data (on writes) and detailed error information in case of errors. This method requires much more hardware but gives more information to devise efficient software solutions for recovery.

Data buffers on the SBus card. The SBus card has two buffers which it uses in a round-robin fashion. It keeps track of the number of packets sent but does not maintain packet context, e.g. destination address. It can only detect that packets are lost or that error responses have arrived. This context throw-away gives limited error handling capability, but safe algorithms are implemented in cluster software to make the transfer reliable.

Error detection might happen late and the information on what went wrong is limited. The benefit of this method is that even with limited resources there can be up to 64 outstanding packets, only limited by the transaction ID field (SCI restriction).

Two buffers are necessary only for performance increase utilizing pipelining, i.e. receiving data from the SBus and at the same time sending an SCI packet from the other buffer.

Streams on the PCI card. On PCI, the requirements are much higher both for context and buffer capacity. A *stream* consists of a data buffer (64 bytes on PSB32, 128 bytes on PSB64) and a context (PCI address, mapped SCI address and status). In order to efficiently utilize the PCI and SCI bandwidth, the streams are capable of doing write gathering and speculative read buffering such that, if many PCI accesses are needed to transfer the data, the PSB will wait for the next expected addresses (and data) and until the buffer is full or empty. Explicit and implicit flush/invalidation of streams can be done based on configuration.

The PCI32 card has 8 write streams and 8 read streams which can be configured in different ways to support different resource allocation configurations based on the PCI address accessing the card. Some streams can be reserved for DMA engine use and/or assigning different number of streams to different address windows within the card's memory range.

The PSB has a complete context of all transactions outstanding on SCI and in progress on PCI. Write gathering can be done on all 8 streams concurrently.

The error handler can get detailed error information on each failing packet (error type, address, and even write data). This way, a single error does not necessarily result in a complete stop of all traffic over the bridge. This is a requirement in I/O applications since one failure otherwise would affect all drivers running the I/O applications.

3.4.3 SCI Packet Requester

The requester builds the SCI packets and keeps track of the packets ensuring that a loss of a packet is detected. When a response arrives it updates the context, transaction ID counter (SBus) or stream status (PCI). The capabilities of the two adapters are quite different.

SBus card SCI requester. The SBus card is designed for efficient writes over SCI and supports only a simple *readsb* command for remote probing operations. The reason for this is found in the write-only programming model used by the Dolphin software and Sun applications. Write posting is more efficient than reading, especially when accessing slow memory (SCI is slow compared to local memory access). A read stalls the I/O bus.

Table 3.1 shows the supported SCI commands on the SBus adapter.

SCI transactions	SBus card	PCI card
writesb	1,2,4,8 bytes	All
readsb	1,2,4,8 bytes	1,2,4 bytes
nwrite64	Yes	Yes
dmovesb	NA	All
dmove64	NA	64 bytes
locksb	NA	4 byte fetch-and-add

Table 3.1. Supported SCI commands

PCI32 card SCI requester. The request packet sender takes the contents of the stream when it is ready and makes the required number of SCI packets to fulfill the work based on the stream contents and configuration. It can only have one outstanding packet per stream at a given time and starts a timer when sending in order to detect lost responses. Each outstanding packet sent from the card has a unique transaction ID that matches the stream number belonging to the packet. Thus, the stream to be updated when the response packet arrives can be determined.

The request channel and the response channel are completely independent in the card. Requests must never block for responses. The PSB32 can have 8 write and 8 read request packets in flight at the same time. The responses can arrive in any order and all errors are handled independently per stream. This method is quite costly in terms of hardware, but in some applications it is necessary.

Stream selection both for write and read transfers are based on a method called *stream combination*. The stream selected (1 through 7 for read and write) by a PCI address addressing the card is determined by the address offset bits. By selecting which address offset bits shall determine the internal 3-bit stream selector, different stream allocations can be set up. This method was developed based on the fact that most transfers are block transfers where

the address is increasing linearly. Using this method several linear data transfers can run in parallel and resource (stream) conflicts can be avoided since the pool of streams can be divided between them. To the user (e.g. linear DMA transfer from another device on PCI) it seems that the stream buffer size increases beyond 64 bytes and that more outstanding packets are possible when streams are combined. The biggest possible combined stream consists of 8 streams giving a virtual buffer size of 512 bytes and 8 outstanding packets.

Read and write streams can be configured individually. The life cycle (when in active use for transfer) of a write stream starts when it is first accessed and is not in use and ends when the corresponding response packet is received from SCI. A read stream life cycle starts when it is accessed and it is not in use and ends when the master on PCI has emptied the fetched data. Both life cycles include a request and response phase on SCI.

The PCI card has equal possibilities for reading and writing. In order to make read performance over SCI acceptable in I/O transfer operations, read prefetching is implemented. This mechanism can be configured to work with 1, 2, 4 and 8 streams. When fully configured (level combination), one read on PCI will result in 8 *nread64* packets on SCI on consecutive addresses. Whenever a stream is emptied (all 64 bytes are read) by a PCI master, a new read request is automatically initiated, always keeping the prefetch pipeline running.

Table 3.1 shows the supported SCI commands on the PCI adapter.

PCI64 card SCI requester. The PCI64 SCI card is very similar to the PCI32 card, but offers more internal resources and enhancements for more efficient use as a SAN interface adapter. The card has 16 read and 16 write streams with increased buffer size (128 bytes). The page size is 4 kByte (or more) and all transfer attributes are stored in the page entry. Associative stream lookup on write streams ensures effective buffer usage and better multiprocessor performance since the stream selection always selects a stream that can be used and is not bound by the static stream combination scheme. In addition an enhanced store barrier (see 3.8.2) per page is included.

3.5 SCI Responder

The SCI card also accepts request packets and converts the packets to I/O bus transactions. This operation is more straightforward since there is no inbound mapping on the current cards. Inbound mapping is the ability to perform address mapping (which also might include address protection) also when a packet arrives at its destination node.

A proper response to the request is sent after the fate of the request packet is determined. Many things might go wrong in the responder, like I/O bus retry time-out, request time-out, bus error, bus parity error, and access protection violation.

Stomped packets or CRC error packets coming from SCI are skipped resulting in a time-out at the sender. Such packets cannot be trusted in any way. Full store-and-forward of request and response packets is therefore used in the cards, i.e., the packet is checked for integrity before it proceeds to the I/O bus.

The SBus and PCI32 cards have capacity for two incoming request packets. This is sufficient for keeping full speed on the I/O bus. The PCI64 has a much bigger input buffer with space for 32 *nwrite64* packets. This will help to keep packets out of the interconnect to avoid retries on the SCI link and in the SCI switches. Increased overall system performance is the result.

3.5.1 Mailbox

The Dolphin SCI cards have also implemented a feature called mailbox. Special SCI packets are tagged (by the address mapping) and handled in the card in a special way. The packets are stored in a location determined by software at the target node (i.e. not using the address offset in the SCI packet header). When a mailbox packet arrives, an interrupt is raised. The benefit of this feature is to enable the drivers on the different cards talk to each other without having access to the address of a remote message buffer.

Mailbox on the SBus card. Inbound packets with address offset bit 47 set to 1 (done by the mapping) will be stored in the mailbox buffer in the card. The driver will be interrupted and will inspect the packet.

Mailbox on the PCI card. These packets are tagged with a special target node ID and stored in a ring buffer in main memory maintained by the card. The driver reads the head of the buffer when triggered and inspects the packets.

3.5.2 Access Protection

The PCI card has also implemented access protection based on address ranges in the 4 GByte address space (Read Only, Read Write and No Access).

It is also possible to enable access protection from groups of source IDs (16 in each group). Any request from these node IDs will result in an error response.

3.5.3 Atomic Access

SCI defines lock operations using *locksb* packets where both operand and data is part of the request packet. The response packet contains the value of the lock location before the operation takes place (old value). The operation, i.e. reading the old value, computing the new value, and writing the new value is executed atomically in the responder.

The PCI card is able to handle 4-byte *locksb* packet requests. The *locksb* action is a read-modify-write operation. Note that the operation is only atomic as seen from SCI. As a consequence, all accesses to synchronization variables must be done from SCI (not locally). The PCI card is capable of using the #LOCK signal, which is rarely used by computer host bridges (CPU/memory-to-PCI bridge).

A fetch-and-add-1 request can be directly generated from the address mapping by the SCI requester. Both read and writes will generate *locksb* with *fetch-and-add 1*. The old value returned on PCI reads will proceed to PCI as a normal read. On PCI writes, the old data is skipped.

3.5.4 Host Bridge Capabilities

To operate as a host bridge-type device the PCI card must also be able to handle the extra tasks required by a host bridge.

Remote configuration. The PCI card has the possibility to do configuration cycles as master. This makes it possible to do remote configuration of e.g. nodes without CPU.

Event reporting. A special unit on the PCI card is implemented to forward interrupts and SERR# from the local PCI bus to a selected host connected to SCI. The card is capable of reporting interrupts on INTA/B/C/D.

I/O cycles. The card is capable of generating I/O read/write commands as a master on the PCI bus. This is necessary to control devices that only implement I/O space access for the control registers (most devices are memory mapped, though).

3.6 DMA Transfers

In order to off-load the CPU for large block transfers, the SCI cards have a built-in DMA engine capable of chaining control blocks. The DMA engine operates concurrently with other traffic both in the requester and responder.

3.6.1 DMA Transfers on the SBus Card

The control blocks are stored in on-board SRAM and the machine is capable of writing to SCI only. The block size has a 64-byte granularity.

3.6.2 DMA Transfers on the PCI Card

The DMA control blocks are stored in a 4-kByte PCI memory block. The DMA engine will fetch the new control block after finishing the current one. Interrupts can be set after any desired block and the engine will stop immediately after an error. The block size granularity is 4 bytes and a 4-byte alignment is required.

3.7 Interrupter

The cards are able to generate interrupts by asserting the interrupt pin. The interrupt handler can investigate the source of the interrupt by inspecting registers on the card. Interrupts are typically used to forward notifications from remote nodes or to notify local errors of any kind.

On the PCI card, there are several ways a node can generate interrupts on remote nodes. Interrupts can be used e.g. to notify end of message transfers.

3.8 Concurrency Issues

When the card is used for message passing as an SAN SCI adapter, there are several issues that are important for the overall performance. Raw bandwidth is one thing, but another thing is how to make efficient use of this bandwidth in a modern multiprocessor computer system.

3.8.1 Write Assembly

A typical use of the PCI card is in an Intel-based server with up to four CPUs. When all four CPUs are using SCI for message passing and there is no synchronization between the processes (or threads) running on the CPUs that regulates access to the card, there might be four different messages being sent at the same time (assuming writes). In this case, the PCI host bridge will not be able to make long bursts for each of the messages. The card must be able to handle four different burst streams and write-gather the data to make efficient 64-byte transfers on SCI.

This can be done on the PCI32 card by dividing the available eight streams into four windows and assign each CPU to one of the windows. The problem with this solution is that the utilization of the streams will be poor and that the CPUs cannot reach peak bandwidth. Combining all streams into one window is also possible but will result in low performance when more than one CPU is accessing the card. The PCI64 card has a 16-way associative stream lookup to avoid this problem. This means that write gathering is possible on all 16 streams concurrently with no restrictions on addresses.

3.8.2 Efficient Store Barrier

Another issue in the scenario above is the store barrier. Each thread (CPU) must be able to check that a sent message really was delivered without any errors. This operation (the store barrier) should be able to be performed without affecting the other sending (writing) CPUs. A typical store barrier in the PCI32 card (and in principle in the SBus card) looks like this:

1. Flush all write data (a register read).

2. Wait until all outstanding write requests have completed (an *ordered* register read will not return data until all outstanding write packets are done).
3. Check if any errors occurred (a read of a status register).

For a single CPU this is an efficient algorithm, but when a CPU is performing a store barrier in the middle of the data transfer of another CPU, the latter transfer is affected for two reasons.

The flush might lead to non-optimal SCI transfers (i.e. *writesb* instead of *nwrite64*) which will slow down the transfer. In addition, the ordered read will stall the packet-sending pipeline for a short time (typically around the latency time). When there are small messages being sent, the above issues will degrade the throughput considerably if store barriers are needed on each message.

The PCI64 card has addressed both these issues by providing a store barrier that avoids the global effects of the write flush and pipeline stall. Each page can be flushed individually and the ordered read is replaced with a simple poll not affecting the other pages/CPUs (except for traffic on the PCI bus).

3.9 Performance

Table 3.2 shows some best case bandwidth numbers for the SCI cards. All numbers are real numbers measured in applications (except for the PCI64 card which has been simulated).

Transfer	SBus card	PCI32 card	PCI64 card
SCI card to memory	30	88	125
Bus master to SCI card	30	104	172
DMA push (Wr)	27	74	125
DMA pull (Rd)	NA	64	125

Table 3.2. SCI cards bandwidth comparison (MByte/s)

3.10 Applications and Topologies

The Dolphin SCI cards are useful for many applications. The technology provides a communication channel with high bandwidth and low latency (2.3 μs) that supports both mapped memory accesses (e.g. CPU to SCI) and DMA transfers. The PCI card also provides additional features that broadens the usability of the technology into the System Area Network (SAN) space.

3.10.1 SAN Interface Adapter

This is currently the main application of the SCI cards. Most cluster applications are using message passing as a means of process communication. SCI is very well suited for this type of communication. Several message passing libraries for the cards have been developed (e.g. MPI [9] and VI Architecture [10]) supporting parallel computing and highly available parallel database servers like Oracle's Parallel Server (OPS) and IBM's DB2.

Traditional network drivers can also utilize the technology to get more speed even though the software overhead in the protocol stack is still a bottleneck. DLPI (Solaris) and NDIS (Windows NT 4.0) are examples of this type of driver.

3.10.2 Remote I/O Connection and Data Acquisition

The PCI cards can also be used for this type of application when there is no CPU attached to the PCI bus implementing an I/O subsystem supporting many PCI slots using PCI-to-PCI bridges. Local configuration is set up remotely from, e.g., a node connected to SCI. Siemens is using SCI and PCI components from Dolphin for remote I/O access in their midrange servers with support for 144 PCI slots usable from 24 CPUs in a system (system RM 600 [4]).

Siemens is also developing software for Windows NT that enables a standard server to connect to the remote I/O subsystem using the PCI card as the host adapter in the server.

Dolphin has also a PMC (PCI small form factor [6]) version of PCI32 to be used typically in data acquisition applications.

3.10.3 Switches and Topologies

SCI can be connected in many topologies. Dolphin has concentrated its work on switch-based topologies since this gives the best resilience and availability required by e.g. database cluster applications. A basic adapter card has only one link controller restricting the topologies to single rings and switches. Several adapters may be used in one machine for redundancy and/or more bandwidth.

A 4-port non-cascadeable switch is available both for SBus and PCI cards (see Figure 3.4. The new switch from Dolphin is cascadeable up to 16 nodes suitable for bigger SAN systems requiring high bandwidth and low latency (PCI cards only). Next-generation switches featuring LC-3 will have up to 32 ports with increased internal bandwidth and lower cost per port.

A special version of the PCI32 card has been developed where there are two LC-2 chips (with a mezzanine connection) per card giving the possibility to make a 2D mesh structure or counter-rotating rings, for instance. SCALI

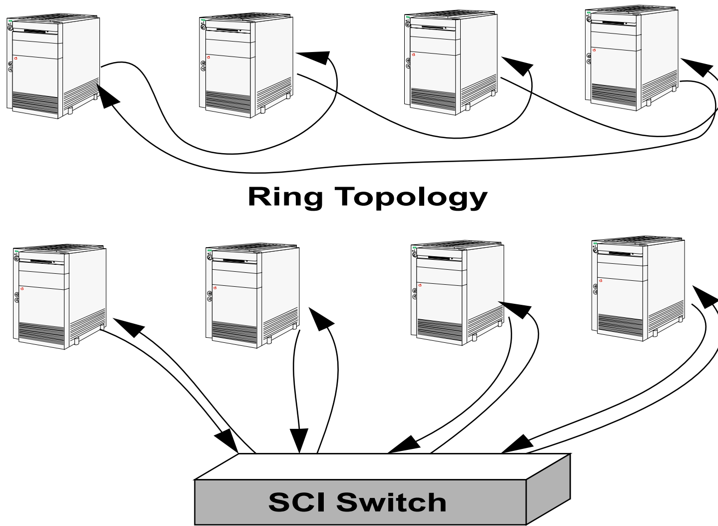


Fig. 3.4. Switch and ring configuration

Computers [5] are using this feature in their installations, e.g. in the large-scale University of Paderborn machine which has 96 nodes connected as a 8 x 12 2D torus [3]. Figure 3.5 shows two ways of connecting the PCI-SCI cards using double SCI links.

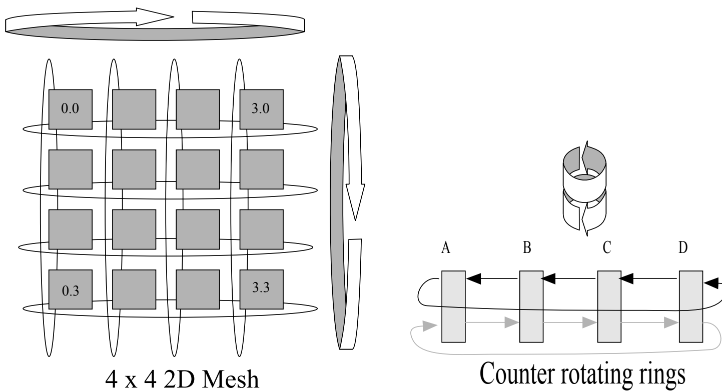


Fig. 3.5. 2D mesh and counter rotating ring configuration

The PCI64 card has the possibility to have two extra LC-2 chips (attached on a mezzanine card) making it possible to construct 3D meshes. The table routing capabilities of LC-2 gives many possible topologies.

The SCI link connection uses standard parallel copper cables and signaling as specified in the SCI subspecification IEEE 1596.8. The links run at 100 MHz. A parallel optical solution exists for the PCI cards and switches (PAROLI, developed by Siemens [4]). This type of connection allows distances of up to 150 meter.

3.11 Cluster Software

To allow easy access to SCI and clustering, Dolphin provides drivers and programming interfaces on various levels. Figure 3.6 shows the different components of the cluster software.

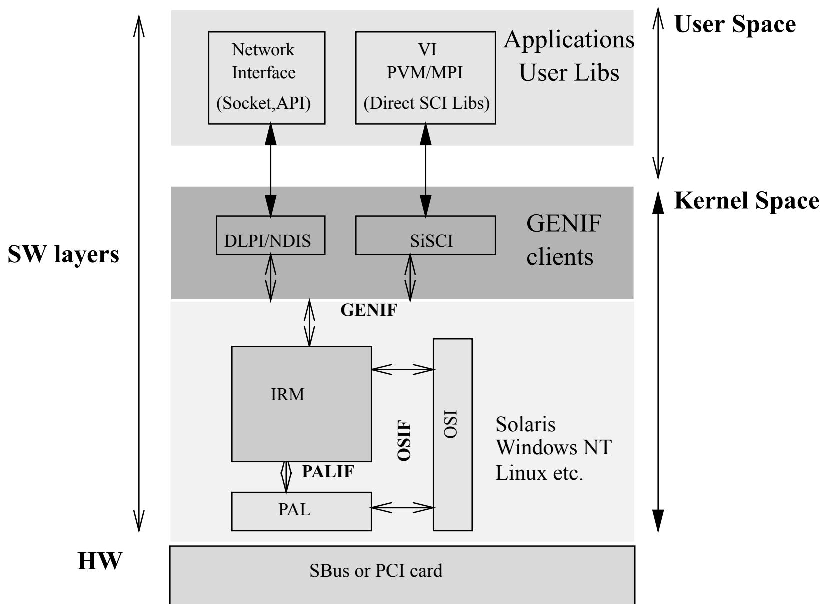


Fig. 3.6. Overview of Dolphin cluster software components

The Physical Abstraction Layer (PAL) is the lowest layer in the software stack. This interface provides a procedural interface for directly programming the hardware independent of the implementation details for the various SCI adapters. Currently there exist PAL libraries for both the PCI and the SBus cards. On top of the PAL is a portable kernel agent called the Interconnect

Resource Manager (IRM). The IRM exports an interface that allows the construction of intermediate device drivers. This GENeric InterFace (GENIF) features a rich set of mechanisms for managing and utilizing clusters. The IRM is particularly suited for clients that need to achieve high availability in mission critical applications. Features such as reliable data delivery, hardware fault detection and isolation, redundancy, and serviceability are supported through the GENIF interface without loss of performance or latency. GENIF has many benefits: it removes the complexity of writing hardware dependent device drivers, and it allows multiple clients to utilize SCI and to coexist in the same computer even running on the same SCI adapter card.

Typical GENIF clients are traditional networking drivers like NDIS and DLPI and more specialized direct SCI interface drivers. These drivers are required to enable applications to access the cluster.

A new interface called SISCI (Software Infrastructure for SCI) has been defined in an ongoing ESPRIT project [7]. Dolphin has made a portable implementation of this interface. The SISCI interface enables an easy-to-use user-level interface to clustering. The portable IRM and SISCI software are currently available on Windows NT x86 and Alpha, Solaris x86 and SPARC, Linux x86, Lynx x86, and PowerPC.

The most common usage of the SISCI interface is the implementation of efficient versions of known libraries/programming interfaces like PVM, MPI, OPS, WinSock2, and VI Architecture [10]. For instance, Scali is using MPI as the message-passing library in their computers.

The main programming features provided on the SISCI and GENIF interfaces include:

- Construction of SCI accessible memory segments.
- Mapping of local or remote memory segments.
- DMA handling and queuing.
- Registration and triggering of remote interrupts.
- Atomic operations and conditional interrupts.
- Barrier and fail detection operations.

References

1. Dolphin Interconnect Solutions. *LC-1/2 Functional Specification.* http://www.dolphinics.no.
2. Dolphin Interconnect Solutions. *PSB32/64 Functional Specification.* http://www.dolphinics.no.
3. University of Paderborn, Paderborn Center for Parallel Computing. *Primergy High Scalable Server.* http://www.uni-paderborn.de/pc2/systems/psc.
4. Siemens AG. *RM600 Enterprise Servers.* http://www.siemens.de/servers/rm/rm_us/rm600e.htm.
5. Scali Computer. *Scalable Computer Systems and Technology.* http://www.scali.com.

6. IEEE Std-1386. *IEEE Standard 1386 Common Mezzanine Card Family CMC.* The Institute of Electrical and Electronics Engineers, Inc.
7. ESPRIT Project No. 23174. *SISCI – Standard Software Infrastructures for SCI-based Parallel Systems.* http://www.parallab.uib.no/projects/sisci.
8. *PVM – Parallel Virtual Machine.* http://www.epm.ornl.gov/pvm/pvm_home.html.
9. *Message Passing Interface Forum.* http://www.mpi-forum.org.
10. *Virtual Interface Architecture.* http://www.viarch.org.

4. The TUM PCI/SCI Adapter

Georg Acher, Wolfgang Karl, Markus Leberecht

Lehrstuhl für Rechnertechnik und Rechnerorganisation – LRR
Institut für Informatik der Technischen Universität München
80290 München, Germany
email: {acher, karlw, leberech}@in.tum.de
http://wwwbode.informatik.tu-muenchen.de/

4.1 Introduction

The SMiLE project (Shared Memory in a LAN-like Environment) at LRR-TUM investigates high-performance cluster computing based on SCI interconnect technology [9]. The major research goal is to develop concepts, programming models, and tools for the efficient use of a distributed shared memory (DSM) system with NUMA (non-uniform memory access) characteristics. One focus within the project is to adapt and optimize "standard" parallel processing software for SCI-based cluster environments. Several communication libraries [8], a POSIX compliant distributed thread package [17], and the MuSE runtime system combining DSM, multi-threading and dataflow techniques [13] have been developed and implemented making use of SCI's hardware-based DSM.

A second focus are hardware developments which include a PCI/SCI adapter [1, 2] and an SCI hardware monitor [11]. The TUM PCI/SCI adapter is targeted for bridging the PC's PCI local bus to the SCI network and serves as basis of the SMiLE PC cluster. Pentium PCs equipped with these PCI/SCI adapters are connected in a cluster of computing nodes with NUMA characteristics.

The lack of commercially available SCI interface hardware for PCs during the initiation period of the SMiLE project led to the development of our own PCI/SCI hardware. Additionally, our own PCI/SCI adapter is designed for extensibility and adaptability. It allows to attach a hardware monitor which is able to deliver detailed information about the run-time and communication behavior to tools for performance evaluation and debugging [12].

The PCI/SCI adapter has to translate PCI bus operations into SCI transactions, and vice versa. The interface to the PCI local bus is implemented by the PLX bus master chip PCI 9060 [15], the SCI side is handled by the Link Controller LC-1 manufactured by Dolphin Interconnect Solutions [4]. However, the control and protocol processing between these two commercial ASICs has been realized with two field programmable gate array chips (FPGAs).

This chapter presents the rationale and design of the TUM PCI/SCI adapter. Especially, the implementation of the control and protocol processing

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 89-101, 1999
© Springer-Verlag Berlin Heidelberg 1999

units will be described in detail. Our approach for a structured hardware design is based on microprogramming concepts enabling us to manage the complex protocol processing.

4.2 The PCI/SCI Adapter Architecture

As already mentioned, SCI has been chosen as the network fabric for the SMiLE PC cluster. Nodes within an SCI-based system are interconnected via point-to-point links in ring-like arrangements or are attached to switches. For the communication between SCI nodes, the logical layer of the SCI specification defines packet-switched protocols. An SCI split transaction requires a request packet to be sent from one SCI node to another node with a response packet in reply to it. This allows several transactions to be overlapped, thus hiding remote access latencies.

For the SMiLE PC cluster, the PCI/SCI adapter serves as the interface between the PCI local bus and the SCI network. As shown in Figure 4.1, the PCI/SCI adapter is divided into three logical parts: the PCI unit, the Dual-Ported RAM (DPR), and the SCI unit.

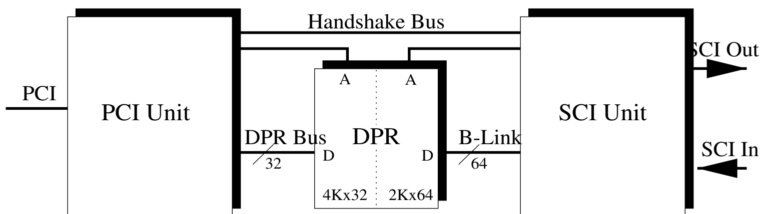


Fig. 4.1. PCI/SCI adapter architecture

The PCI unit (Figure 4.2) interfaces to the PCI local bus. A 64 MByte address window on the PCI bus allows to intercept processor-memory transactions which are then translated into SCI transactions. For the interface to the PCI bus, the PCI9060 PCI bus master chip from PLX Technology is used [15]. The main task of this chip is to transform PCI operations into bus operations following the Intel i960 bus protocol, and vice versa. Two independent bi-directional DMA channels are integrated, allowing direct memory transfers between PCI and SCI initiated by the PCI/SCI adapter.

The protocol conversion from the i960 bus to SCI involves an address translation to access specific pages on remote nodes. The address translation data is stored in the address translation cache (ATC, 8k*24 bits), allowing a maximum of 8192 pages with 4 kByte each to be addressed. For economic reasons, only an 8-bit node address and a remote address range of 512 MByte are used, so only three ATC RAMs are necessary for the implementation.

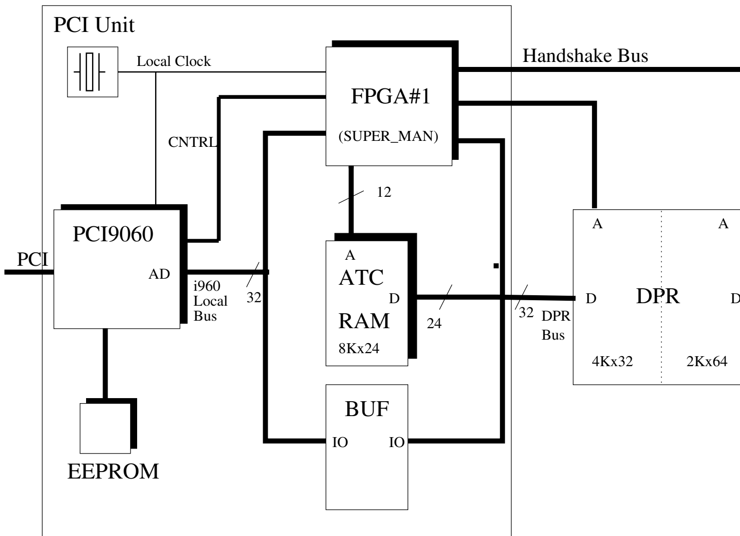


Fig. 4.2. Block diagram of the PCI unit

The packets to be sent via SCI are buffered within the Dual-Ported RAM (DPR). It contains frames for outgoing packets allowing one outstanding read transaction, 16 outstanding write transactions, 16 outstanding messaging transactions using a special DMA engine for long data transfers between nodes, and one read-modify-write transaction which can be used for synchronization. Additionally, 64 incoming packets can be buffered. During the initialization of the DPR, constant values are written to the frames of the outgoing SCI packets. Composing an SCI packet then only requires variable data like addresses or data to be written into the appropriate fields.

The SCI unit interfaces to the SCI network and performs the SCI protocol processing for packets in both directions. This interface is implemented with the Link Controller LC-1, which implements the physical layer and parts of the logical layer of the SCI specification. A 64 bit-wide synchronous back-side bus for SCI link chips, the B-Link [6], connects the SCI unit and the DPR. The B-Link data format comprises the SCI packet data format and some additional information to accomplish a simpler handling.

The PCI unit and the SCI unit both contain an FPGA. These chips perform the control and protocol processing within and between the two units. As FPGA resources are rather limited compared to ASICs, an implementation based on an FPGA requires some special design rules to be considered, especially when targeting higher frequencies. Due to the rather complex task of translating PCI transactions into B-Link packets and vice versa, the implementation of the PCI unit's FPGA demanded a careful application of those rules. The following section gives a detailed description of the design aspects for the PCI unit's FPGA, discussing the design problems and their solutions.

We then describe briefly the much less complex SCI unit and finally present some performance data of our PCI/SCI adapter implementation.

4.3 SCI Packet Encoding and Decoding

4.3.1 Overview of Packet Processing

To illustrate the various actions during packet processing, the following example describes a remote write operation.

The PCI9060 transforms the PCI write transaction into the PCI unit's local bus operation. The PCI unit controller recognizes that an SCI write transaction has to be composed. With the help of the ATC, the PCI address will be translated into the address for the remote SCI node. The translated address will then be written into the DPR. Subsequently, the buffered data will also be written into the DPR. The target address and the number of bytes to be written are used to generate the SCI command for single-byte writes.

The PCI unit controller informs the SCI unit controller about the completion and the position of the request packet within the DPR. Simultaneously, a timer is started which stops upon receipt of the response packet. This feature allows packet loss to be identified.

The SCI unit controller starts the transmission of the packet to the LC-1 via the B-Link. The LC-1 directs the packet to its output link, thus sending the packet to the remote SCI node.

On the receiver SCI node, the LC-1 directs the packet onto the B-Link. The SCI unit controller writes the packet into an incoming packet frame within the DPR and informs the PCI unit controller about the arrival of the packet. The PCI unit controller extracts the SCI command from the packet in order to determine the actions to be performed. The address will be extracted from the packet's address field and put onto the internal i960 bus. Afterwards, the data will be put on the i960 bus and written into the memory by the PCI9060. After completion, a response packet will be generated on the receiver node and sent back to the requester node, where the timer will be stopped and the packet frame released.

4.3.2 Choosing the Technology

The previous description indicates that despite the use of two specialized commercial ASICs, the main task of the PCI/SCI bridge has to be done by custom hardware: the conversion from the bus-oriented i960 protocol to the packet based B-Link protocol of the LC-1 and vice versa.

It became clear that this task cannot be easily achieved with lower integrated programmable logic devices (CPLD technology). The PCI unit has to handle reads and writes from the PCI side as well as read and write requests

from the SCI unit. Furthermore, the implementation of a semaphore/lock transaction and an efficient DMA block transfer were considered to be very useful additions. In order to fully use the low SCI latency, none of these transactions (except the lock) should need any additional software actions for packet encoding and decoding. Thus all encoding/decoding has to be implemented in hardware. An SCI lock transaction cannot be transparently handled through the PCI bus and the whole SCI network, because the locked PCI bus transfers are more general than the SCI lock operations.

Eventually, the usage of SRAM based XC4000E-series FPGAs from Xilinx [18] seemed to be the only way to implement the controlling parts of the PCI/SCI bridge. Not only their relatively low cost and good performance, but also their reconfigurability in the environment of 'unknown' chips (the PCI9060 and the LC-1 were used the first time) lead to this decision. This may also decrease the amount of work to implement new features in the future. The VHDL [16] implementation tools from Synopsys were chosen for the design entry.

During the design phase, the RAM functionality (implementing small RAM blocks inside the FPGA) of the XC4000E became very useful, because many details could be implemented in an easy and straightforward way (namely by the micro-sequencer described below).

In the following, the FPGA of the PCI unit is called the 'SCI Upload/Packet Encoder Management' (SUPER_MAN).

4.3.3 Internal Structure of the FPGA

The SUPER_MAN FPGA chip is responsible for controlling and coordinating the translation of read/write accesses into B-Link packets, and vice versa. It comprises several functional units which are shown in Figure 4.3.

While evaluating the specifications for the controller, it became obvious that the somewhat complex data flows and the encoding/decoding actions in the PCI unit could be separated into five main parts (objects, see Figure 4.3), communicating only with a few signals (commands).

The functional units of SUPER_MAN are controlled by the Packet Manager unit (PAC_MAN). It contains a microcode sequencer and a writable control store with the microprograms controlling the other functional units.

For example, there are microprograms for translating a PCI address into an SCI address and writing them into the DPR, for generating SCI commands from addresses or byte enables, and for transmitting data into the DPR.

The Local Bus MANagement unit (LB_MAN) recognizes whether there is a request on the local i960 bus. Depending on the state of the read/write and DMA signals, it determines which microprogram has to be executed. The LB_MAN sends the appropriate control information to the TRANSAction Management unit (TRANS_AM). This unit checks its transaction queues for outstanding requests or free entries and forwards the received control information to the PAC_MAN. If an SCI packet arrives, a control command for

the PAC_MAN will be generated from the SCI command using a translation table.

The TRANS_AM determines the DPR base address for the appropriate packet frame. It also coordinates between local bus and SCI requests and communicates with the corresponding unit within the SCI unit.

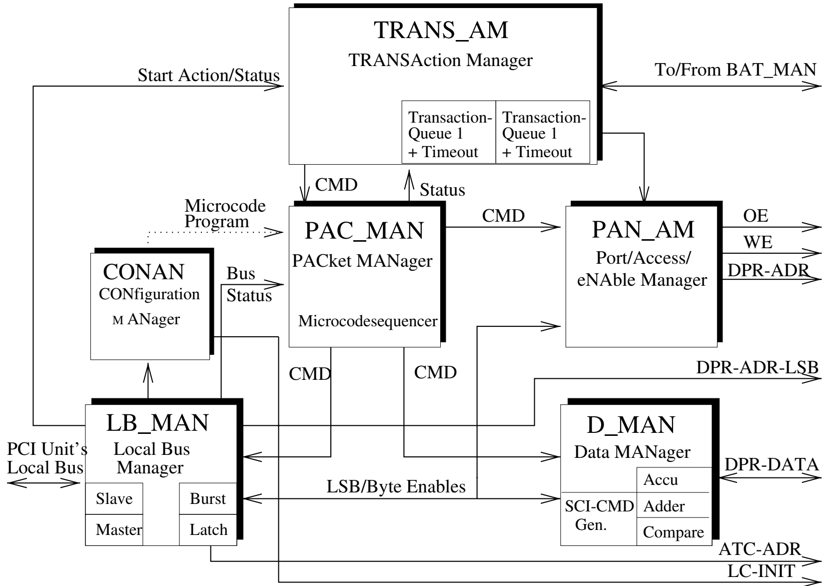


Fig. 4.3. Structure of SUPER_MAN

The five separate units of the SUPER_MAN FPGA are now described in detail:

– **Local Bus Manager (LB_MAN)**:
This controller serves the i960 local bus. This includes accepting accesses via the PCI9060 (PCI→SCI), generating a status describing the access type, and starting accesses to the PCI9060 (SCI→PCI, issued by a specific command such as READ or WRITE). Although controlling the i960 bus state, LB_MAN has no knowledge of the transferred data itself or any data flow concerning other buses than the i960 bus.
– **Data Manager (D_MAN)**:
This unit interfaces to the external DPR bus. Depending on the command, it stores internally some fields of the PCI and SCI data or generates specific data itself (e.g. the SCI commands). It is also used as an ALU for the 'Bounded Add' operation (needed for the SCI lock transaction).
– **Port/Access/Enable Manager (PAN_AM)**:
Encoding and decoding involves the transfer of data from different sources (i960 bus, incoming packet data in the DPR, address translation RAM

and the data generated by D_MAN) to various sinks (i960 bus, outgoing packet data in the DPR, internal registers). Although there seems to be a great number of these data flows (complicated by different bit fields and offsets) they can be reduced to 15 simple flows. Depending on the command, PAN_AM sets the needed internal and external buffer enable and read/write signals, allowing addressing of some specific locations (bit fields, like the SCI command field) in the 'current' B-Link packet (packet base) and combining them with a source or sink.

– **Transaction Management (TRANS_AM)**:
  The Transaction Management sets the DPR packet base address corresponding to the current LB_MAN status (read/write from PCI) or to the next incoming packet. An arbiter selects between both possibilities. In some cases it is necessary to switch to another packet base (e.g. get read data from a read response), so there are specific packet base 'override' commands. As the SCI network supports split transactions, there are also two bookkeeping units for regular writes and DMA writes with 16 entries each. Since PCI only supports posted writes, SCI read transactions cannot overlap on the initiator side.
  The main task of TRANS_AM is starting the appropriate encoding/decoding action, depending on the status of the i960 bus and the command from an incoming SCI packet. The actions themselves are controlled by the next unit.
– **Packet Manager (PAC_MAN)**:
  In order to encode or decode a packet and to serve bus accesses, this unit issues the right sequence of commands to the other four units. Depending on the needed action, this sequence can be linear, with wait states or a few branches.

4.3.4 Structure of the Packet Manager as a Microcode Sequencer

As stated above, the various actions for packet encoding and decoding are controlled and coordinated by the Packet Manager. Its behavior can be described as a finite state machine (FSM) consisting of about 60 states. Each state outputs the four 4-bit commands to the other functional units. In total there are more than 80 state transitions, most of them unconditional. Yet, some transitions are more complicated, e.g. wait states or the initial dispatching of the 16 needed actions (e.g. generating single-byte write requests).

The first approach in implementing this state machine was a conventional one, using the `case-when` construct in VHDL. After the first tests with only 30 states, it became evident that this way was not feasible for an FPGA. The highly irregular structure of the state decoding and the relatively wide (16 bits) and densely packed command output were synthesized into a very large and slow circuit (max. 14 MHz for an XC4013E-3). Because the target frequency was 25 MHz, this result was considered unacceptable.

Although it would have been possible to use a special tool for the FSM synthesis, the simple command scheme of the other functional units and the RAM capability of the FPGAs led to another idea: the implementation of a small, specialized micro-programmable processor. This approach was also stimulated by the circumstance that the program-like flow of the actions could very easily be translated into microcode.

The usage of microcode for a program sequence is not new, but mainly used in microprocessor applications. Older FPGAs without the RAM capability only had the possibility to implement microcode with external RAM (slow) or with lookup tables (ROM). For each change in the microcode, the design would have to be re-routed. Thus microcode in FPGAs is rare.

A detailed analysis showed that a very simple microcode sequencer could cover all needed state transitions and output signals, some of them without any need of explicit or complex implementation (see Figure 4.4):

The internal structure of the microcode sequencer consists of the following parts:

– **Normal execution**:
  Comparable to some other microcode machines (e.g. the AM29xx series [3]), the 'Next-PC' is encoded in the microinstruction word. Hence, there is no need for an adder and an additional branch logic. The necessary RAM for the 'Next-PC' is arranged as 64*6 bits. The XC4000-series has 32*1-bit RAM cells, accounting for 12 configurable logic blocks (CLBs) for the RAM itself and another 6 CLBs for the additional demultiplexers and flip-flops for the clocked operation.

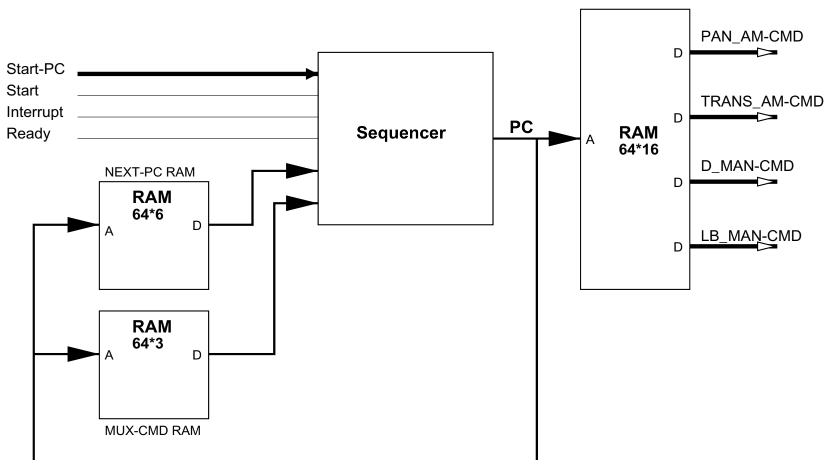


Fig. 4.4. Structure of the microcode sequencer

- **Command outputs**:
  Parallel to the PC RAM input, there are the four 64*4-bit RAMs for command generation. Their clocked output feeds the four other units, so in each microprogram step the units can operate in parallel.
- **Starting a program**:
  For all encoding/decoding, TRANS_AM sets a 4-bit command describing one of the 16 possible actions. In order to start, the microcode sequencer jumps to one of the first 16 locations in the control store where each action has its entry point.
- **Changes in the program flow**
  - **Wait states**:
    When a packet is encoded or decoded, the state machine has to wait until all data is accepted or delivered by the PCI interface, or a response comes back from the SCI link.
  - **Interrupt and Return**:
    If a DMA write is aborted by the PCI9060, the unfinished packet must not be sent until **all** 64 bytes are written into the packet. The next DMA cycle has to return to that microcode position where the aborted cycle left off (interrupt).
- **Programming control**:
  There is an additional multiplexer input for the CLB-RAM address, allowing the simple reconfiguration of the microcode (not shown in Figure 4.4).

4.3.5 Microcode Examples

To show the ease of development for the chosen microcode architecture, the generation of a single-byte read request (R_RQO) is shown. This textual description is processed by a simple parser, generating the binary patterns for programming.

PC	Next-PC	PAN_AM	D_MAN	TRANS_AM	LB_MAN	MUX_CMD
2	19	TRANSADR_2_DPR	GEN_ADR_LSB	SEND_R_RQO	NONE	GO
19	20	TIDCMD_2_DPR	GEN_RRQ_CMD	NONE	NONE	GO
20	21	NONE	NONE	SET_RRSI	NONE	MUX_WAIT
21	63	DPR_2_DATAL	NONE	SET_RRSI	DATA_READY	GO
63	0	NONE	NONE	DONE	DONE	GO

The control flow is started with command '2', thus the sequencer begins at PC=2. In this step the translated address is written into the request packet. Due to synchronization stages on the B-link control logic, the packet can be marked 'ready to send', even when it is not completely assembled. In the second step the translated target ID (TID) and the read request command (RRQ) are written into the appropriate places. At PC=20, the execution waits until the response packet with the needed data arrives, then the packet address is set to the incoming packet (SET_RRSI) and the buffers are opened

to the local bus. At this point, the waiting PCI9060 access can be finished (DATA_READY) and in the last step all actions are finished.

The most complex microprogram is the execution of a lock operation at the receiving node (only the 'Bounded Add' of the SCI specification is implemented). It consists of 13 steps, implementing the read/modify/write cycle with the arguments sent, plus the additional response packet generation:

PC	Next-PC	PAN_AM	D_MAN	TRANS_AM	LB_MAN	MUX_CMD
/* Start the read cycle with a locked bus*/						
7	41	DPR_2_ADRGEN	NONE	NONE	NONE	GO
41	42	DPR_2_ADRGEN	LOAD_ADRLSB	NONE	NONE	GO
42	43	DPR_2_ADRGEN	NONE	SET_RSO	READ_LOCK	GO
/* Now wait for the data and write it in the response packet*/						
43	44	DATAL_2_DPR	NONE	SET_RSO	NONE	MUX_WAIT
/* Also load the data in the ALU*/						
44	45	NONE	LOAD_DATA	NONE	NONE	GO
/* Read upper bound */						
45	46	MAX_2_FPGA	NONE	NONE	NONE	GO
/* Read increment, latch upper bound */						
46	47	OP_2_FPGA	LOAD_COMP	NONE	NONE	GO
/*Latch increment and perform bounded add, start write back */						
47	48	DPR_2_ADRGEN	LOAD_ADD	NONE	WRITE	GO
/*Wait until data is accepted */						
48	29	FPGA_2_DATAL	STORE_DATA	SET_RSO	NONE	MUX_WAIT
/*Assemble response packet, swap source and target ID, early send */						
29	30	TIDCMD_2_DPR	GEN_R16_CMD	SEND_RSO	NONE	GO
30	31	SIDTRID_2_FPGA	NONE	SET_RSO	NONE	GO
31	32	FPGA_2_TIDTRID	LOAD_DATA	SET_RSO	NONE	GO
32	0	FPGA_2_TRID1	STORE_DATA	DONE_RSO	NONE	GO
/* Done... */						

This code re-uses some parts of the response-outgoing (RSO) code part from the normal read action. Although the code has to deal with pipelining effects (e.g. D_MAN load must occur one cycle after its corresponding PAN_AM read enable), the microcode approach shows its full elegance.

4.3.6 Benefits of the Micro Sequencer

While developing and testing the prototype, the reconfigurability of the microcode (without a new VHDL synthesis) was heavily used for debugging. Nearly all unknown or uncertain behavior of the two ASICs could be analyzed. The robustness of the various data paths (setup/hold) could be tested with specific test (stress) patterns and sequences generated by the microcode.

After getting the prototype to work, some (previously unplanned) features were implemented just by altering the microcode. In the same manner, some activities were tuned saving one or two clock cycles. Some other features presently not implemented such as support for remote enqueuing for an improvement of the Active Messages communication layer [10] and a touch-load (reducing the effective read latency) should require only minimal changes in the hardware itself.

In this application the microcode sequencer design shows its superiority over the conventional way of designing state machines, especially for the PCI/SCI protocol transactions with many unconditional transitions. Additionally, the possibility of reprogramming without a change in the synthesized hardware makes it very attractive for prototyping and debugging.

4.4 The SCI Unit

The SCI unit (Figure 4.5) is connected to the DPR via the B-Link. The B-Link Access and Transaction MANager (BAT_MAN) controls B-Link arbitration, reading or writing the DPR (on the SCI unit side), and drives the B-Link control signals. It is also implemented on a FPGA chip, but is much less complex than the SUPER_MAN FPGA.

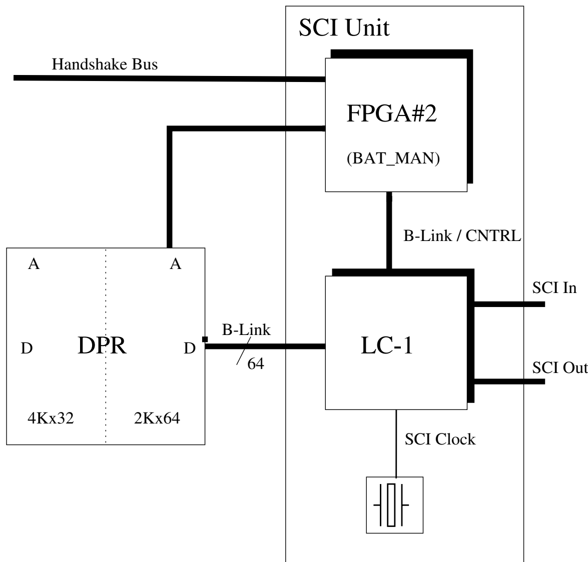


Fig. 4.5. Block diagram of the SCI Unit

4.5 Preliminary Results for the PCI/SCI Adapter

The logic of SUPER_MAN currently requires about 325 CLBs (56% of the XC4013 CLB resources), thus leaving enough room for new features. The FPGA of the simple SCI unit consist of a XC4005E-2 (about 53% CLB usage). Both units (and therefore the i960 bus and the B-Link) can run with 25 MHz. The SCI link of the LC-1 is clocked with 50 MHz, transmitting data on both edges.

The connection of two Pentium133-based machines via two PCI/SCI bridges showed block transfer rates of up to 40 MByte/s (CPU-initiated i960-DMA transfer and *nwrite64* packets) and a bandwidth of 4 MByte/s for regular (non-burst) shared memory writes over SCI, demonstrating the overhead

of the protocol conversions. The PCI-to-PCI latency for these writes (on an unloaded SCI network) is about 2.7μs, comparable to other PCI/SCI adapter cards.

4.6 Related Work

The PCI/SCI adapter developed at CERN [14] uses also the LC-1 as the SCI link controller and an FPGA from AT&T. Additionally, Dolphin now offers PCI/SCI adapters [7, 5] based on the LC-1 and its successor LC-2. The board implements all encoding and decoding logic in an ASIC. Due to the 'write-combining' feature and its custom PCI unit, it achieves higher data rates (70-80 MByte/s) even for non-DMA transactions.

4.7 Conclusion

In this chapter, we described in detail the design aspects and implementation of the PCI/SCI adapter for the SMiLE PC cluster. Additionally, we demonstrated that, by means of traditional microprogramming concepts instead of complex state machines, the FPGA design can be simplified. New features of the PCI/SCI adapter can be integrated with minimal hardware changes, mainly by adding microcode.

The PCI/SCI adapter allows the attachment of a hardware monitor which gathers detailed information about the run-time and communication behavior of a program. The experience gained with the development of the PCI/SCI hardware will help us in designing and implementing the hardware monitor. The hardware monitor is described in Chapter 24.

References

1. G. Acher, H. Hellwagner, W. Karl, and M. Leberecht. A PCI-SCI Bridge for Building a PC Cluster with Distributed Shared Memory. In *Proceedings Sixth International Workshop on SCI-based High-Performance Low-Cost Computing*, pages 1–8, Santa Clara, CA, Sept. 1996. SCIzzL.
2. G. Acher, W. Karl, and M. Leberecht. PCI-SCI-Protocol Translations: Applying Microprogrammable Concepts to FPGA. In R. Hartenstein and A. Keevallik, editors, *8th International Workshop on Field Programmable Logic and Applications, FPL'98*, volume 1482 of *Lecture Notes in Computer Science*, pages 99–108, Tallinn, Estonia, Aug. 1998. Springer-Verlag.
3. Advanced Micro Devices. *Bipolar Microprocessor Logic and Interface: Am2900 Family Data Book*, 1985.
4. Dolphin Interconnect Solutions. *Link Controller LC-1 Specification, Rev.1.06*, 1995.
5. Dolphin Interconnect Solutions, 1998. http://www.dolphinics.com.

6. Dolphin Interconnect Solutions. *A Backside Link (B-Link) for Scalable Coherent Interface (SCI) Nodes*, 1996.
7. Dolphin Interconnect Solutions. *PCI-SCI Adapter Card Functional Overview*, Jan. 1999. Version 2.1.
8. M. Eberl, H. Hellwagner, W. Karl, M. Leberecht, and J. Weidendorfer. Fast Communication Libraries on an SCI Cluster. In H. Hellwagner and A. Reinefeld, editors, *Scalable Coherent Interface: Technology and Applications. Proceedings of SCI Europe'98*, pages 165–175. Cheshire Henbury Tamwoth House, P.O. Box 103 Macclesfield SK11 8UW, UK, 1998.
9. H. Hellwagner, W. Karl, and M. Leberecht. Enabling a PC Cluster for High-Performance Computing. *SPEEDUP Journal*, 11(1), June 1997.
10. H. Hellwagner, W. Karl, and M. Leberecht. Fast Communication Mechanisms–Coupling Hardware Distributed Shared Memory and User-Level Messaging. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, Nevada, June 30–July 3 1997.
11. R. Hockauf, W. Karl, M. Leberecht, M. Oberhuber, and M. Wagner. Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In D. Pritchard and J. Reeve, editors, *Euro-Par'98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1-4, 1998 Proceedings*, volume 1470 of *Lecture Notes in Computer Science*, Berlin, Sept. 1998. Springer Verlag.
12. W. Karl, M. Leberecht, and M. Oberhuber. Enforcing Deterministic Execution of Parallel Programs – Debugging Support Through the SMiLE Monitoring Approach. In H. Hellwagner and A. Reinefeld, editors, *Scalable Coherent Interface: Technology and Applications. Proceedings of SCI Europe'98*, pages 83–90. Cheshire Henbury Tamwoth House, P.O. Box 103 Macclesfield SK11 8UW, UK, 1998.
13. M. Leberecht. A Concept for a Multithreaded Scheduling Environment. In F. Hoßfeld, E. Maehle, and E. W. Mayr, editors, *Proceedings of the 4th Workshop on PASA'96 Parallel Systems & Algorithms*, pages 161–175. World Scientific, 1996.
14. H. Müller, A. Bogaerts, C. Fernandes, L. McCulloch, and P. Werner. A PCI-SCI Bridge for High Rate Data Acquisition Architectures at LHC. In *Proceedings of PCI'95 Conference*, pages 156 ff, Santa Clara, USA, 1995.
15. PLX Technology Inc., 625 Clyde Avenue, Mountain View, CA. *PCI 9060 PCI Bus Master Interface Chip for Adapters and Embedded Systems*, Apr. 1995. Data Sheet.
16. R. Lipsett et. al. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 1990.
17. M. Schulz. SISCI Pthreads: SMP-like Programming on an SCI Cluster. In *High-Performance Computing and Networking (Proc. HPCN Europe)*, volume 1401 of *LNCS*, pages 566–575. Springer-Verlag, 1998.
18. Xilinx Inc., 1995. http://www.xilinx.com.

Part III

Interconnection Networks with SCI

Apart from the raw hardware capabilities of the adapter cards, low-level routing policies and the configuration of the network are both critical for the overall system performance. Such aspects are covered in this part.

First, Chapter 5 gives a comprehensive description of low-level SCI protocols, including the important topics of packet formats, flow control, and bandwidth multiplexing. With this background, the architecture of current switches is analyzed and an improved rule-based routing scheme is proposed as a general routing method for high-speed networks.

The SCINET project described in Chapter 6 investigates SCI in the context of large-scale data acquisition systems in fusion reactor experiments. Again, switches play a decisive role in the efficient utilization of the network and it is shown how commercial switches can be better exploited. The authors have performed extensive simulations to underpin their approach. Moreover, the authors argue that Banyan topologies often provide better throughput and latency than conventional multi-stage networks.

The following two chapters are especially useful for those who are faced with the task of determining the optimum configuration of a large SCI network with as few hardware components (switches) as possible. While this is a non-trivial optimization problem with many restrictions, analytical models can be developed to provide adequate approximations to the solution, as shown in Chapters 7 and 8.

How many nodes can be attached to a single SCI ringlet without imposing too much network contention? According to the mathematical model in Chapter 7, about ten nodes should be used as a maximum. Clearly, this is a crude estimate, and it depends very much on the limited data injection rate of the current PCI bus hardware. Several other factors, like message locality, bypass rate, packet size, etc. also contribute to the number of nodes that can be reasonably served on a single ringlet.

Chapter 8 also deals with scalability, but this time with respect to the optimum number and configuration of ringlets in large systems. The results of the analytical model in Chapter 8 is used by Scali in the configuration of their large systems, like the one installed in Paderborn. Based on Dolphin adapter

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 103-104, 1999
© Springer-Verlag Berlin Heidelberg 1999

cards that are equipped with more than just one link controller, it is possible to build multi-dimensional tori. Chapter 8 shows how many dimensions are needed for a given system size.

5. Low-Level SCI Protocols and Their Application to Flexible Switches

Andreas C. Döring, Wolfgang Obelöer, Gunther Lustig, Erik Maehle

Medical University of Lübeck
Institute of Computer Engineering
Ratzeburger Allee 160, D-23538 Lübeck, Germany
email: {doering,obeloeer,lustig,maehle}@iti.mu-luebeck.de
http://www.iti.mu-luebeck.de

5.1 Introduction

The purpose of SCI is to provide a fast interconnection technology for up to thousands of components. This chapter discusses several aspects of the protocols used in SCI, especially regarding requirements and possibilities for their implementation in hardware. One motivation behind the SCI standardization effort was to enable SCI as interconnect for peripherals or memory subsystems. Therefore, we discuss how the protocols can be implemented efficiently and in a small chip area.

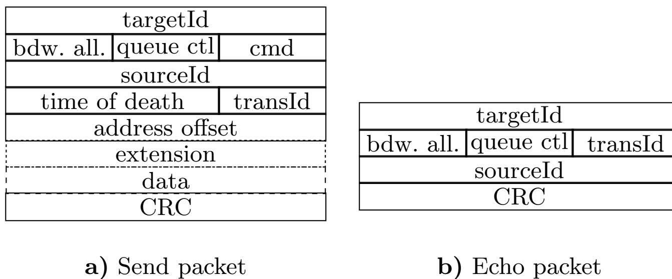
The first part deals with the data format of SCI packets (Section 5.2), the protocols for allocating buffer space (Section 5.3) and bandwidth (Section 5.4). In the second part, the architecture of current switches is taken as a starting point for elaborating further demands and consequences for switch architectures. One particular design question (strip-off decision) is discussed in detail and an implementation option is given. A major issue of this chapter is the introduction of the rule-based description method for routing algorithms in Section 5.7. It is not only used to describe the examples of the strip-off decision but it reveals a general approach to implementing routing methods for high-speed networks.

5.2 Data Format of SCI Packets

In SCI, data is defined in terms of symbols, which are 16-bit words, i. e. transmission is always done in multiples of 16 bits. Of course, larger entities equipped with address, priority and so on are needed for transmission through the network. These units are typically called packets. Their size is a critical factor. If the communication system supports a wide range of packet sizes, the bandwidth offered by the physical layer can be used very effectively by transmitting almost exactly the required amount of data. However, handling packets with a variable size is harder compared to fixed size, especially in

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 105-123, 1999
© Springer-Verlag Berlin Heidelberg 1999

hardware. The choice of buffer size is also determined by the switching technique, e. g. packet or worm-hole switching (see [1]). For example, in packet-switched networks like SCI all buffers must be able to host the largest packets. Since SCI aims at low-latency memory access and at cheap implementation, it supports pay load data sizes of 0, 16, 64, and optional 256 bytes. These sizes are oriented at typical cache line dimensions and the need for small synchronization packets. In order to detect transmission errors, the packets are equipped with a 16-bit CRC checksum (Cyclic Redundancy Check). It does not cover the bits for the low-level protocols to avoid re-computation of the checksum at every node. Overall, the packets can have the following formats (Figure 5.1). If not otherwise mentioned, the fields correspond to one symbol (2 bytes).



a) Send packet **b)** Echo packet

Fig. 5.1. Data format of SCI packets

There are two different kinds of packets with a common structure, send packet and echo packet:

- Destination address (*targetId*).
- Fields for the flow control and bandwidth allocation protocols (*bdw. all., queue ctl* (4 bits each)).
- Command field (*cmd* (8 bits)) used to distinguish the packet types.
- Identification of the source node (*sourceId*).
- *Time of death*: This is the time when a packet is to be deleted, because it has consumed too much time. The node where it has been generated will eventually repeat it, because it has not received the echo in time. Of course, this is only found in non-echo packets.
- Transaction number (*transId*).
- *Address offset*: 48-bit address of the referenced object at the target node.
- *Header extension* (optional, 16 bytes): May be used for advanced protocols. When using the coherency option some of these fields are defined by the standard.
- *Data*: Depending on the transaction type, this field contains read or to-be-written data of the referenced object, e. g. a cache line. Depending on the application it may have a different size.

– *CRC:* Checksum as explained before.

If no packet is transmitted over the link, idle symbols are used to fill this gap. Although these symbols do not carry data, they are used for the link protocols described in the subsequent sections. The detailed format of an idle symbol is given in Section 5.4. Though the packet formats appear to be complicated, they can be processed easily in hardware. One currently available chip contains about 25000 gates (or 100000 transistors, without buffers) which is not very much for today's circuits.

5.3 Flow Control

A high speed communication system should avoid the loss of data. One reason for data loss is node overflow, i. e. more data is sent to a node than it can consume or store. The mechanism to avoid this situation is called flow control. It can be done in several ways.

One way is *end-to-end* operation. By acknowledging the receipt of each packet and re-sending a packet after a time-out when the acknowledgment is missing, a reliable transmission can be established. In the network, packets may have to be discarded due to buffer overflow. Using large buffers at every node in the network reduces the probability of overflow and thus the probability of re-sends. However, the hardware cost for the buffers would be high, especially in a large network. In addition it is hard to provide fairness between nodes with different distances to a common destination. A worst case scenario would be that no messages from certain nodes arrive in a heavily loaded region.

The second option which is used for instance in Myrinet [4] and HIC [17] is flow control on the link level, also called *back-pressure*. This means that each link is equipped with backward control which will stop the transmission if the data cannot be stored or processed anymore. For bi-directional connections this can be accomplished by inserting control characters in the data stream of the opposite direction. Because there is some delay in the transmission and processing of flow control information, a certain amount of buffer memory must be reserved. Furthermore, additional delay is enforced because the information about available buffer space has to be carried backward.

5.3.1 Flow Control in Rings

SCI applies a mixture of both methods. Flow control is only applied between two nodes of a ring, while the visited nodes between them just pass data. For this reason the transmission rate of an outgoing link has to be as high as that of the incoming one.

Assume a node (requester) wants to send data to another node (responder) in the same ring. If the requester allocated buffer space in the responder

before sending the data, a high delay would be unavoidable. Thus, the data is sent without guarantee that it can be absorbed by the destination node (a 'trial' in the following explanation). Therefore sending packets on the ring is sometimes unsuccessful because packets are dropped.

For one responder there may be several requesters, and in one ring there are usually several responders at a time[1]. In order to give all requesters the same chance of being served, SCI applies a queue allocation protocol for flow control. The flow control fields in request, response, and echo packets (see Figure 5.1) are used to exchange information between requester and responder, i. e. to inform the requester about the success of the transmission and the state of the responder as well as to encode the kind of trial assigned to a packet transmission. The general sequence is started by the requester transmitting a send packet to the responder, which answers with an echo packet. If such a trial was successful, the requester can remove its copy. Otherwise it re-transmits the packet.

The sequence of transactions is separated into phases by the responder. The protocol makes sure that outstanding requests from one phase are served in the next phase before newly arriving requests are accepted. This has the consequence that only requests from two phases have to be distinguished at one time. To limit the amount of data for the protocol, a coloring scheme is used, i. e. only three different kinds of trials are distinguished: normal, A, and B. The protocol is symmetric regarding A and B.

Each responder is in one of four states in which it accepts normal and A, only A, normal and B, or only B requests, respectively. Thus, for every phase there are two states which improves the protocol in the context of rings with distributed requesters. The responder changes its state when it accepts or rejects a packet, or when a time-out occurs. The responder cycles through these states, i. e. every state has exactly one successor. The protocol only needs to determine when it must switch states.

5.3.2 Packet Sequence in SCI

When a packet is inserted into a ring for the first time, it is marked as a 'normal' transmission. In the responder there may be two reasons for rejecting this packet:

Case 1: There is not enough queue space available.

Case 2: There is queue space available, but there are previous requests which were rejected and have to be served first.

If a normal trial is rejected by a responder in 'only A' state, the packet is re-sent labeled with B. Similarly in 'only B' state, rejected normal trials

[1] It is a bit confusing that these terms are used in the SCI context for the original ends of a transaction which may be located at different rings **and** for transmitting and receiving inside a ring which may be intermediate nodes of the message path. Nevertheless they are applied here to adapt common usage.

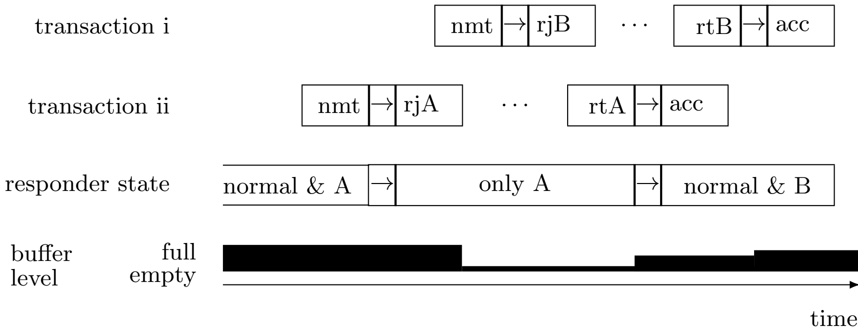


Fig. 5.2. Sample queue allocation sequence at one responder (nmt=normal trial, rjA/rjB = reject A/B, rtA/rtB = retry A/B, acc = accept)

are re-sent as A transmissions. For an example, see Figure 5.2. Assume the responder is in the 'accept normal and A' state, has a full buffer and rejects a normal packet (case 1). Then this request should be favored over all newly arriving normal packets. Therefore, the responder changes its state to accept only state A packets and the repeated request is assigned with state A. In this period, arriving normal packets are assigned phase B and will not be served until all outstanding A requests are done (case 2, transaction i). In this example this is only one request which is served after some time when the buffer is nearly empty. Hereafter, B requests—and also normal requests—are accepted, as for transaction i. As can be seen, the buffer fills up again, because it holds these packets until they are processed. Symmetrically, if in this state a request is rejected, it is assigned state B and the responder only accepts B requests. Thus, if in situations of high traffic load filled-up buffers are common, it will be seldom that a request is accepted at the first time; the responders are most of the time in the accept-only-A and accept-only-B states.

5.3.3 Determination of State Transitions

While changing into the 'only'-states is straight-forward, leaving them is harder. It has to be figured out when all outstanding A or B requests are done. SCI applies a combination of two methods. In most cases it is enough to count how many future A (respectively B) requests are generated by rejecting a normal trial. The result can be compared to the number of served A (or B) requests. Although this approach allows a responder to quickly enter a state of acceptance of normal packets, it has limitations:

- The size of the counter limits the ring size, thus inhibits scaling. As for speed reasons a counter with only a few bits is desirable.
- It bears the danger of misalignment of the counters for the outstanding A or B requests. To avoid repeated CRC computation in the protocols, the

protocol bits are not covered by the checksum. Hence, transmission errors in the protocol bits are not discovered by the CRC.

For example, a responder may count up too far by receiving a corrupted packet. In such a case the responder waits forever for an outstanding request, that does not exist anymore. To avoid this, a time-out mechanism is added. It has to make sure that every outstanding request on the ring has had a chance to be repeated. If during this time for instance no repeated A transaction occurs, no pending A requests are in the ring. This knowledge about the possibility to re-send is represented by sufficient available bandwidth on the ring for a certain time. The protocols are separately executed for request and response packets.

Altogether the fairness achieved is not total, but the probability that a message gets a large delay because it is discriminated too often is diminishingly small. It is obvious that the time critical operation of accepting/rejecting a message cannot be much simpler; the flow control field (2 bits) from the header has to be checked according to one of four states at the responder. Changing the state in the responder is similarly easy and less time critical: checking whether the A/B-counter has hit zero, or a time-out has been reached. This time-out is not generated by a clock but by a special method in the bandwidth allocation scheme, presented in the next section.

5.4 Bandwidth Multiplexing

By separating the transmission of requests from their serving, the two potential bottlenecks in the system can be handled separately: overloaded servers and overloaded media. While the *queue allocation* protocol handles the first problem, dividing the medium capacity among all packets is the task of the *bandwidth allocation* protocol. In non-ring systems with shared medium, e. g. Ethernet, this is done by a distributed media access protocol. After an allocation conflict, both transmitters have to wait for a minimum randomized delay before a next try is allowed. This method wastes a lot of bandwidth, because there are gaps when the medium is not used. In media with only one driving source at one point, bandwidth multiplexing can be done much better, because the demands for bandwidth are known beforehand. A good example is ATM which applies certain traffic-shaping methods to guarantee quality of service [22].

5.4.1 Bandwidth Management in One Ring

On a ring the situation is a mixture of the two approaches mentioned. The ring segments are single-sourced, but the demands for ring bandwidth are distributed. Locally, only the packets from the incoming link and the local requests (packets in the send queue) can be seen. A new message can only be

injected if bandwidth on the ring is available. A purely local decision cannot avoid the starvation of a node, due to the lack of knowledge about other requests in the ring.

The problem of *ring access* is as follows. If a message A of a particular size has to be injected into the ring and currently the ring is free, it is impossible to know how long this will last. The insertion of a message with a certain number of symbols into the ring is the replacement of idle symbols by data symbols. If an insertion has started and a message arrives from the incoming link, this second message is delayed in a so called bypass buffer. In order to ensure that no message on the ring is destroyed by overflow of the bypass buffer, message insertion is only possible if the bypass buffer is empty. In order to send a message, a node has to wait for this situation; it starts transmission in the next available bandwidth section.

The *bandwidth allocation* protocol ensures that no node has to wait too long until it can start transmitting, by setting a further restriction. Its aim is to favor a subset of more important transactions over others while still guaranteeing the progress of all packets. Since the priorization adds complexity to the protocols, it is optional. A node which can inject messages with higher priority is called an 'unfair-capable' node. Nodes which are only 'fair-capable' inject messages with the lowest priority, thus guaranteeing that messages with higher priority can get more bandwidth. Obviously the importance of messages is best known where they are generated. That is why every message is equipped at its source with a 2-bit priority stored in the control field. Thus, four different priorities are possible. Bandwidth should be divided equally among messages with the same priority. That means on one ring, all messages at the top of the queue with the same priority have the same expected waiting time until they are transmitted.

Dividing the bandwidth among four priority classes on the ring would require a rather complicated protocol. To avoid it, on the ring only two priority *classes* are used, high and low. The mapping of the message priority to the priority class on the ring is done by every node. It maintains an approximation of the current maximum ring priority (a value in the range from 0 to 3). If a message in this node has the same or higher priority than this approximation, the message is assigned to the high priority class, otherwise to the low one. Clearly the protocol can be separated into two parts: bandwidth allocation according to the priority classes, and estimation of the ring priority.

Bandwidth allocation: On the ring a lot of nodes can transmit messages concurrently. That is bandwidth is not a single global resource but several transmissions can take place concurrently. On the ring a free bandwidth section is given by idle symbols.

If a node can access the ring, i. e. its bypass buffer is empty, the allocation protocol further restricts the permission of a node to insert a message. For fairness, the node may have to be forced to transmit the idle symbols

it receives to the following nodes keeping the bandwidth for them. Hence, protocol-related information is encoded in the idle characters.

5.4.2 Idle Symbols

To allow priorization, two kinds of idle symbols are distinguished: high-type and low-type. High-type symbols may only be deleted for allocation by a message in the high priority class. However, the transmission of high-type idles is delayed in a node with non-empty bypass queue, that is the occurrence of high-type symbols is saved until the FIFO has run empty. Following low-type idles are replaced by high-type idles, conserving the fraction of high-priority bandwidth. In the same manner low- and high-priority idles are restored when a packet is stripped off the ring. There is no precise value for the fraction of bandwidth for low-priority transactions, because this would require a distributed accounting scheme. By avoiding this, all the necessary information can be encoded in the idle symbols.

To distinguish between the two priority classes, the node has to maintain an *estimation of the ring priority.* This is the highest priority value among the messages in the ring or with the intention to enter it. It is easy to figure out when the estimation of the ring priority has to be increased. By taking the maximum priority (0 to 3) of all observed packets in a ring or bypassing a node, an approximation can be transmitted in idles (alias for idle symbols) and stored in the nodes. A more difficult problem is decreasing this estimation, especially when removing a packet with the actual ring priority. Then, it is not clear to which value the ring priority should be reduced. Possibly more packets with the same high priority are present. The solutions based on the fact that the combination of a packet and its echo passes all nodes in the ring. Together, all present requests can be observed. Therefore, in the header of every packet the maximum priority of all passed nodes is computed. This result is put into the echo; in another field of the echo the maximum priority of all nodes the echo passes is computed. If the echo is stripped off the ring, it has an up-to-date estimation of the ring priority which is put into the following idles until a new packet or echo passes this node.

In summary, the *idle symbols* consist of the following fields:

estimate of ring priority	allo-cation count	circu-lation count	low-go	high-go	type	age

Since they are too short to be protected by a CRC checksum like all other packets, they consist of two halves where one is the Boolean negation of the other.

The low-go and high-go bits characterize free bandwidth sections. A certain amount of idles has to be present on the ring without offering allocatable bandwidth to allow compensation of differing link speeds. These are distinguished by the type bit. The remaining bits are described later.

5.4.3 Time-Out Determination

The protocols for bandwidth allocation and queue allocation are not totally independent of each other. As pointed out, the bandwidth allocation protocol provides a time-out criterion for the queue allocation. It works as follows. A responding node needs the information that all other nodes in the ring have had a chance to transmit a request. This is equivalent to the fact that an idle with allocatable bandwidth has completed a round.

The method applied in SCI to check this consists of extremely simple elementary operations, yet it is effective. The 'allocation count' bit contained in every idle symbol is inverted in one selected node, the 'scrubber'. Hence, there is always[2] at least one point in the ring where two adjacent idles (may be separated by data packets) have different allocation count bits. Of course this pair cycles through the ring and can be easily observed in every node. The nodes store the most recently received allocation count bit. If they are not allowed to send packets into the ring—regardless whether there is a waiting packet or not—the node puts the stored allocation count bit into the idle symbols ignoring the currently received one. In this manner, any differences in the following allocation counts are destroyed. These differences are used by a responder to generate the time-out for outstanding state A or state B requests. To really ensure this, the state will only be left if four such changes are observed.

In the same way the 'circulation count' allows detection of missing echoes; it gives nodes the possibility to know when the time a symbol needs to make one round on the ring has passed. Since an echo packet is transmitted in place of the request or response packet it belongs to, a lost echo packet can be identified securely.

Altogether the protocols can be implemented by relatively inexpensive building blocks. In hardware terms the most expensive parts are the buffers (receive and send buffers, at least two of them are necessary, and bypass buffer). Since these parts are found in every SCI application it is reasonable to implement them as one design, e.g. an integrated circuit. The demands on such a unit go beyond the link protocols and will therefore be explained separately in the next section.

5.5 Network Interface

SCI interfaces can be applied to a wide range of applications, e. g. memory controllers or host interfaces. There is a considerable common hardware effort for implementing the physical interface, the buffers and protocol engines, justifying the design or production of a separate unit. This has happened on

[2] There may be one cycle where this is not true, but in the next step the property is fulfilled again.

the market in the past few years and a vital development of chips and design macros can be observed. Though driven by the high-end computer market, a migration to the intended low-cost region can be observed. In this section the general requirements and existing products of basic building blocks for making SCI interfaces are discussed.

5.5.1 Requirements

The basic ring operation requires the existence of the bypass buffers. It is obvious from the flow control protocols that every node needs at least a receive or a send buffer. For nodes which act as requesters as well as responders, separate send and receive buffers for both types of packets are needed, i. e. altogether at least four buffers. Of course all the finite automata for the protocols (see Sections 5.3 and 5.4) have to be implemented, but they require a comparatively small hardware effort.

An especially critical aspect is the physical interface. It incorporates the line drivers and receivers which are amplifiers with a precisely defined delay and electrical behavior. For the receiver the crossing of a clock boundary (received clock to internal clock used also for transmission) at a very high rate must be implemented.

An aspect on the ring-side of the interface that has not yet been covered is addressing. As noted in Section 5.2, SCI packets carry a 16-bit node address ('target id') of their destination. For the network interface of the destination it is therefore simple to decide whether a certain packet is aimed for it or not. In this case, the node knows that it is the responder with regard to this packet and it can apply the according protocols. For a network interface in a switch this time-critical decision is harder to make. Whether a packet has to be taken off the ring depends on its further path through the network. This decision is even harder for echo packets, because the network interface has to consume those echo packets that belong to packets it has injected into the ring. To ensure this, the targetId and the transactionId of the echo have to be checked. This operation is equivalent to a lookup in a table addressed by the contents, i. e. an associative memory (sometimes abbreviated CAM).

It is interesting to see in which way existing products have implemented the SCI standard. A brief overview is given in the next subsection.

5.5.2 Products

Meanwhile a certain number of chips or design macros are available, some of them are sold only as part of systems like the cache controllers in the Data General series [6]. Though some of the designs are already in the third generation (e.g. Dolphin LC-2 [11]), there has been no one supporting all SCI features, e. g. 'unfair-capability'. A striking fact is the used technology and the performance that has been reached. While the first hardware implementation of parts of an SCI interface (a test chip by IBM) required 0.5

μm BiCMOS technology, the latest Dolphin/LSI chips can be produced in a standard CMOS process (0.5 μm, 500 MByte/s). This demonstrates the growing design experience for SCI specific problems. Sequent [26] uses a GaAs device manufactured by Vitesse [27] with a link bandwidth of 1 GByte/s. The Interconnect Systems Solutions (ISS) SCI LinCChip-8 concentrates on low system cost by applying only 8 data wire pairs (10 wire pairs total) and a high transmission clock. ISS also sells an ASIC macro for integration into custom chips. With a 0.5 μm CMOS process, ISS claims to reach a link speed of 100 MByte/s.

It is interesting to observe a common feature of all known SCI interface units: Their backside interface is a 64-bit wide bi-directional bus. This is in no way implied by the SCI standard. But the requirements for high bandwidth and moderate frequency at the connections and the restriction of packaging (LC-2 has a 225 pin ball grid array packet) have led to this common solution.

The SCI interface units are valuable for the construction of stand-alone SCI components, for network adapters, or for switches. However, the requirements for routers are special and strongly influence the overall performance of an SCI system. Therefore, the next section will go into detail on this topic.

5.6 Routers

An SCI router (synonym for switch here) is a network internal node that connects two or more rings or routers. It affects the speed of the network, its versatility, expandability and the usability in real-time systems, or multicomputers with coherent distributed shared memory. Therefore, the design of routers bears a lot of challenges.

5.6.1 Requirements

The demands on networks imply demands on routers. Especially the basic expectation that the network transports all packets within a finite time requires the handling of problems like deadlocks or lifelocks. A substantial part of network latency and network bandwidth is contributed by routers. In detail the demands are:

- The time from the entry of a packet (first bit arrives) at one interface until it starts leaving the router (again first bit) on the next ring is called *router latency*. Similarly, the time a packet needs to pass the router when it stays in the same ring is the *bypass latency*. These terms are distinguished because the second figure is normally much smaller. Both should be as low as possible.
- The second important measure is the overall bandwidth that can be supported by the switch. It is the total amount of data that can flow through the router simultaneously. This figure has to be large as it limits system performance in high-load situations.

- A problem of networks with switches in general are *deadlocks*. A deadlock is characterized by a set of messages which cyclically wait for each other on common resources like buffers or links. For SCI this means that some routers are not able to empty their receive queues since the packets cannot be transported to the next ring because all send buffers are full. If the progress within this next ring depends on the router under consideration, none of the affected messages will ever proceed. To avoid this, special means for controlling the buffer dependencies are needed.
- Another issue is reliable operation of the network. The router should be robust against transient errors like misguided packets and the failure of other routers or rings in the network. This property refers to *fault tolerance* and implies methods to adapt to fault situations by guiding messages on detours to their destinations.
- An additional property concerns the application for example in vehicles or airplanes: *real-time behavior*. This means that some messages have a strict time limit concerning their arrival. An extension of the SCI standard for this class of problems is under development.
- The performance of the network can be considerably improved if the choice among several usable paths incorporates the load situation of parts of the network (*adaptivity*). In this manner the traffic can be balanced and local overloads ("hot spots") can be resolved quickly or can be avoided. This requires capturing, distributing, and evaluating of load information.
- If the routers themselves support *cache coherency protocols*, shared memory systems may scale much better. This gave reason to consider extending the standard [21].
- For optimizing application mapping, data about the load distribution of the network and the performance of the individual rings are highly desirable. This implies that the routers have to be able to measure actual load and transmit this information to an appropriate node (*performance monitoring*).
- In workstation clusters or in automation systems the requirements change frequently. To meet the special demands by different message occurrence patterns and changing network structure, *flexibility* of the router is required.

How far these requirements are met is mainly determined by the architecture of the router.

5.6.2 Products and Challenges

Information on existing SCI routers is rare, currently there seems to be no one-chip solution on the market. For economical reasons the routers consist of several integrated circuits. Among them are the SCI interface chips described in Section 5.5.2.

Figure 5.3 illustrates the logical architecture of a router [13]. It consists of several network interface chips connected in a bus-like structure. This solution is supported by the backside interface of the used Dolphin chip [2] already containing bus arbitration logic. Thus, no further hardware is required. The frontside interface is attached to the ring and uses bypass FIFOs. This architecture leads to a low price but limits the router degree. This is caused by the central bus which cannot provide enough bandwidth for a large number of ring interfaces.

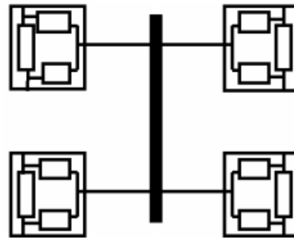


Fig. 5.3. Structure of a simple SCI switch

Using other central connection methods, e. g. a crossbar, overcomes this bandwidth restriction. However, the use of standard SCI interface chips still restricts the power and flexibility of the router because of the separation of the ring interfaces.

In comparison to other network technologies, SCI routers have a remarkably high bandwidth and low latency. For small networks the choice of an appropriate topology is not critical. Deadlock avoidance is provided by fixed path determination, thus excluding adaptivity. Cache coherence, real-time traffic support, performance monitoring, and fault tolerance are not supported.

5.6.3 Flexible Router

For future systems the routers have to fulfill a wide range of requirements. Even the individual applications may put different demands on the network in the same system. But in many applications the structure of the system changes frequently. In such an environment a fixed router would only serve some applications well. For many other applications important demands like time bounds or bandwidth will not be met even with such a high-performance physical layer like the one of SCI.

To avoid the construction of many different routers for these demands a router is needed that can be programmed to suit a broad range of applications. Clearly, as it cannot execute a sequential program to route a message, a much faster method is required.

From the complicated and large area of design aspects for flexible routers only some (strip-off decision, data path, main routing decision) are shown and for two of them solutions are discussed (Sections 5.6.4 and 5.7).

5.6.4 Strip-off Decision

As pointed out in the introduction of the network interfaces (Section 5.5), an important problem for the design of the router is the decision whether to strip off a packet or not. The models implemented so far use either some kind of mask and compare operation or a table.

Mask and compare (Figure 5.4) extracts a part of the 16-bit target address `targetId` and compares it to a given value. Current implementations do this several times (four times in the example) to gain more flexibility. The configuration of the example routing decision consists of 16-bit registers mask[i] and compare[i] (i=0..3). In hardware the expression is fast and requires little effort: it is just a special notation of the widely used PALs, where the individual rows in Figure 5.4 are the product terms.

```
Strip_off :=  ((targetId  AND  mask[0]) = compare[0])  OR
              ((targetId  AND  mask[1]) = compare[1])  OR
              ((targetId  AND  mask[2]) = compare[2])  OR
              ((targetId  AND  mask[3]) = compare[3])  ;
```

Fig. 5.4. Mask and compare operation

To find out whether a packet should leave the ring at this router, table look-up is another possible method. Since a table with direct access from the target address would be rather large, a selection of parts of it, e. g. high and low byte, is applied. That means a table of 256 bits is accessed with one byte of the targetId; the table content specifies whether the packet is to be taken off the ring. Though this increases the range of topologies, the strip-off decision supports all networks with less than 257 nodes and some others with more nodes.

In order to support additional features like fault tolerance or adaptivity, the decision has to be much more powerful. Hence, the flexibility of this operation strongly influences the overall capabilities of the router.

If the router or a ring breaks, the ring topology normally loses its regularity. Of course not all messages are affected. Thus the function for strip-off for a given message is complicated in some nodes, in others it is comparatively simple. To take this into account in the following, a new scheme is introduced. It takes a variable amount of time in terms of clock cycles. Note that the clock in this context is at the rate of the link frequency, e. g. 125 MHz, which means that the individual steps have to be completed in less than 8 ns.

Such a design is a compromise between speed and flexibility. Reflecting a compromise of a low bypass latency and the coverage of a large fraction of practically relevant problems, we propose the following.

Four decisions are started in parallel (Figure 5.5). While A takes only one cycle, B and C have two cycles and the most complicated decision (D) has to be completed after three cycles. The decisions differ in the aspects they consider. The first unit takes only target and source addresses into account, while the second one regards fault-tolerance aspects as well. The third unit considers only adaptivity aspects and packet header information. Finally, the fourth unit has to cover all aspects (fault-tolerance, adaptivity and packet specification). The first three decisions result in one of three outcomes: 'strip packet off the ring', 'leave packet in ring', or 'don't know'. The configuration of the decision units has to ensure that the two simultaneous results of the second and third decision are not contradictory. If one of the faster decisions comes to a result other than 'don't know', the outcome of the others is ignored.

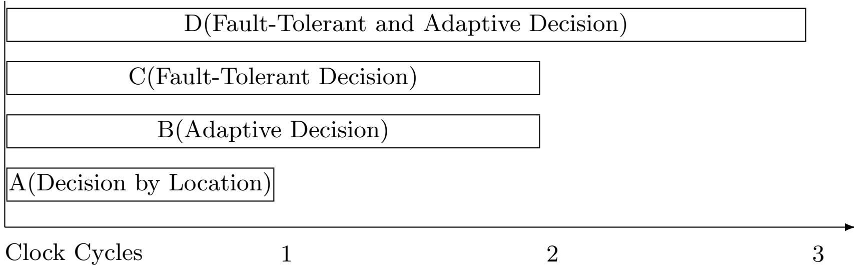


Fig. 5.5. Flexible strip-off decision

Details about the implementation—especially of the decision unit A—can be found in [10].

5.6.5 Routing Decision and Topology

A stripped packet has to be put into an appropriate output ring. That means the router has to determine the next section of the packet's path through the network from the header (targetId, sourceId, command field). Of course, a proper strip-off decision (see Section 5.6.4) is already part of this problem, since it decides for one ring whether it is to be used or not. This strip-off decision only takes one message into account, while the 'main' routing decision in the router has to take care of other messages, at least to avoid deadlocks, to regard priorities (e. g. for real-time behavior), to ensure fair progress for all messages, and for adaptivity. Thus, this decision is quite complicated and has the major burden of the flexibility. The various facets can be divided as follows:

- *Deadlock prevention and fault tolerance* inhibit the use of certain outputs or buffers. In some cases the occupation of output buffers in presence of other messages has to be checked.
- *Purposiveness* (routing towards destination) strongly favors some outputs because they provide progress for a packet.
- *Scheduling* resolves the conflicts for common resources like interface buses.
- *Adaptivity* provides dynamic weights on packets, output links or combinations thereof, giving an approximation as to which decision seems to contribute to a good overall performance. For instance, it may estimate the delay for the transmission of the message to the next appropriate router on the outgoing links from current transfer behavior, link priority, and packet size.
- Furthermore, stale packets in buffers have to be identified and discarded, i. e. the time of death for all packets in buffers has to be observed over time.

These decisions represent complex algorithms, requiring an appropriate description method which is introduced next.

5.7 Rule-Based Routing

As software cannot be used to execute routing algorithms, special hardware is required. These hardware structures have to be configurable in order to implement various routing algorithms. The generation of the configuration data from the high-level algorithm requires an appropriate abstract description method. Since traditional hardware description languages do not offer enough abstraction, a new method is needed. This was the starting point of the RuBIN project (Rule-Based Intelligent Networks). Its main approach is the basis for this section. Originating from knowledge-based systems in the area of artificial intelligence, a rule-based description method is used. The rules used have the simple **IF** `<condition>` **THEN** `<action>` form. This combines the advantages of being abstract, well-defined, intuitive and can be mapped to a hardware structure for fast execution. For a detailed description see [5, 8, 9].

The basic idea is that routing can be characterized by many distinct cases, which can be recognized by few conditions. Each case requires some simple actions that can be carried out quickly. The notion of rules describes these cases well, see [10] for a complete example. One rule looks like this:

```
IF    link_state(1) = ok           AND
      packet.targetId MOD 8 = 1    AND
      out_queue(1) < 2
THEN  insert_packet <- 1;
```

An arriving packet is checked whether the output 1 is in order (`link_state` is ok) and not too heavily loaded (i. e. its output queue contains fewer than two packets) and the packet goes in the right direction (using the lowest three bits of the target id). If all conditions are fulfilled, the rule is applicable and the conclusion is executed. In this case the result instructs the hardware to insert the packet into ring 1.

Every rule uses the keywords `IF` and `THEN` in order to mark the two parts, premise and conclusion. While the premise is a predicate logic expression characterizing the associated case, the conclusion lists all actions for this case. The complete description of an algorithm includes several sets of such rules. For fast execution the rule base is translated into table entries (one per rule). Hence, the processing of a rule base can be done in three steps:

1. Evaluate the basic logic terms from the premises of the rule set in parallel. For typical routing algorithms, e. g. [7], these are between 5 and 15 terms like comparisons and simple arithmetic.
2. Calculate an index into a table for selecting the right rule. This can be done by linear combination, concatenation, etc.
3. Look up the conclusion in a table and execute all its commands in parallel.

Note that no sequential search is performed. Normally the commands are of the kind 'increment a variable' or 'set up a connection for the packet'. In addition there can be more than one rule base. These bases are executed concurrently driven by events like arrival of a packet. This temporal management is also part of the rule-based description.

A tool, namely the rule compiler, is used to translate the abstract description, map it onto the hardware structures and generate the programming data (contents of the table) of the router. A more sophisticated example for the rule based description of the strip-off decision can be found in [10]. To evaluate the performance in hardware, an implementation of a fairly complicated routing algorithm was done using FPGA technology. It turned out that it takes 50 ns. It is clear that the application of ASIC technologies allows a much faster implementation. With the same technology, FPGAs are usually 10 times slower than a dedicated ASIC. However, the speed difference is smaller for integrated memories and they are part of the rule interpreter. For the much simpler strip-off decision we investigated an implementation that reminds of a programmable array logic device (PAL). These devices are currently widely used and reach on-chip delays well under 5 ns.

5.8 Conclusion and Outlook

Flexible and cheap switches are essential for exploiting the benefits of SCI in large network configurations. This chapter presented design and implementation aspects of such switches, which use a rule-based approach. Their

specific requirements were derived from analyzing the SCI protocols. Our rule-based approach combines a high-level description of routing algorithms with an efficient implementation method.

SCI has been an ambitious project from its beginning. Since especially at the time of the standardization process there were no comparable developments on the market, the standardization could apply the best methods for the arising problems regardless of compatibility issues. Only recently a proprietary product with very similar properties and techniques [25] was introduced; however, superior SCI products are available. The visions of the designers [14] have not been fully achieved, yet the demand for a universal fast interconnect for applications like interfacing peripherals to a PC, workstation clustering, or high-end parallel computers is large.

Acknowledgment

This work was sponsored by the German Research Council DFG (MA 1412/3) in cooperation with SFB 376 'Massive Parallelität' at the University of Paderborn, Germany.

References

1. H. Ahmadi, W.E. Denzel. A Survey of Modern High-Performance Switching Techniques. *IEEE Journal on Selected Areas of Communication*, Vol. 7, No. 7, pages 1091–1103, 1989.
2. K. Alnes. *Dolphin's Multiprocessor SCI Interfaces for Clusters and SMP Systems.* Dolphin Interconnect Solutions, 1995.
3. D. Anderson, D. Shanley. *PCI System Architecture.* Addison Wesley, 1995.
4. N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W.-K. Su. Myrinet: A Gigabit-per-Second Local-Area Network. *IEEE Micro*, Vol. 15, No. 1, pages 29–36, 1995.
5. W. Brockmann, T. Kosch, E. Maehle. Rule-based Routing in Massively Parallel Systems. *Proc. 4th Euromicro Workshop on Parallel and Distributed Processing - PDP'96*, pages 154–161, IEEE Computer Society Press, 1996.
6. R. Clark. *SCI Interconnect Chipset and Adapter: Building Large Scale Enterprise Servers with Pentium Pro SHV Nodes.* White Paper, Data General Corporation.
http://www.dg.com/about/html/sci_interconnect_chipset_and_adapter.html.
7. C.M. Cunningham, D. Avresky. Fault-Tolerant Adaptive Routing for Two-Dimensional Meshes. *Proc. First Int. Symposium on High Performance Computing Architecture*, pages 122–131, IEEE Computer Society Press, 1995.
8. A.C. Döring, G. Lustig, W. Obelöer. The Impact of Routing Decision Time on Network Latency. *Proc. 4th PASA Workshop on Parallel Systems and Algorithms*, pages 67–83, World Scientific Publishing, Singapore, 1997.
9. A.C. Döring, G. Lustig, W. Obelöer, E. Maehle. A Flexible Approach for a Fault-Tolerant Router. *Proc. Symposium on Parallel and Distributed Processing – Workshops (Workshop on Fault-Tolerant Parallel and Distributed Systems*

FTPDS 98), Lecture Notes on Computer Science 1388, pages 693–713, Springer Verlag, 1998.
10. A.C. Döring, W. Obelöer, G. Lustig, E. Maehle. Flexibility for SCI-Networks with Rule-Based Routing. *Proc. SCI Europe*, pages 5–11, Cheshire Henbury, UK, 1998.
11. Dolphin Interconnect Solutions. *Link Controller LC-2 Specification.* Data Sheet, Dolphin Interconnect Solutions, 1997.
12. P. Fraigniaud, C. Gaviolle. *Interval Routing Schemes.* Technical Report RR94-04, Laboratoire de l'Informatique du Parallèlisme, Ecole Normale Supèrieure de Lyon, 1994.
13. P. Gustad. A low cost CMOS 500 MByte/sec SCI Link Controller. *Symposium on High Performance Interconnects*, Invited Talk, 1996.
14. D.B. Gustavson. The Scalable Coherent Interface and Other Related Standards Projects. *IEEE Micro*, Vol. 12, No. 1, pages 10–22, 1992.
15. Hewlett-Packard Company, Convex Division. *Exemplar Architecture S-Class and X-Class Servers.* First Edition, 1997.
16. IEEE. *ANSI/IEEE-Std. 1596-1992 Scalable Coherent Interface (SCI).* IEEE, 1993.
17. IEEE. *ANSI/IEEE-Std. 1355-1995 Heterogeneous Interconnect (HIC).* IEEE, 1995.
18. M. Ibel, K.E. Schauser, C.J. Scheimann, M. Weis. High-Performance Cluster Computing Using SCI. *Symposium on High Performance Interconnects*, pages 7–19, 1997.
19. D.V. James, A. Nakamura, A. Ludke, D. Scheel. *Draft—Control and Status Register (CSR) Architecture for Microcomputer Buses.* Available at ftp.scizzl.com/p1212.r/P1212.pdf, 1998.
20. V. Karamcheti, A.A. Chien. Do Faster Routers Imply Faster Communication? *International Workshop on Parallel Computer Routing and Communication*, pages 1–15, Springer Verlag, 1994.
21. S. Kaxiras. Kiloprocessor Extensions to SCI. *Proc. International Parallel Processing Symposium*, pages 166–172, IEEE Computer Society Press, 1996.
22. M. Katevenis, S. Sidiropoulos, C. Courcoubetis. Weighted Round-Robin Multiplexing in a General-Purpose ATM Switch Chip. *IEEE Journal on Selected Areas in Communication*, Vol. 9, No. 8, pages 1265–1279, 1991.
23. SCALI. *The HS Series.* Datasheet, 1997.
24. SCIzzL (David Gustavson). *Compare SCI and ATM, FibreChannel, HIPPI, Serialbus, SerialExpress, SuperHIPPI.* White Paper, Scalable Coherent Interface Local Area MultiProcessor Users, Developers, and Manufacturers Association, 1997. http://www.SCIzzL.com/SCIvsEtc.html.
25. Sebring Systems. *SRC 3266 Sebring Ring Connection.* Data Sheet, Sebring Systems, 1997.
26. Sequent. *Sequent's NUMA-Q SMP Architecture.* 1996.
27. Vitesse Semiconductor Corporation. *Compliant Link Controller 1 GByte/sec SCI VSC7201a.* Datasheet No. G52141-0, Vitesse Semiconductor Corporation, 1996.
28. Bin Wu. *The Applications of the Scalable Coherent Interface in Large Data Acquisition Systems in High Energy Physics.* PhD Thesis, University of Oslo, 1996.

6. SCI Rings, Switches, and Networks for Data Acquisition Systems

Harald Richter[1], Richard Kleber[1], Matthias Ohlenroth[2]

[1] Institut für Informatik, Technische Universität München
D–80290 München, Germany
email: richterh@informatik.tu-muenchen.de
[2] Fakultät für Informatik, Technische Universität Chemnitz-Zwickau
D-09111 Chemnitz, Germany

6.1 Introduction

In plasma-physical fusion devices, the ionized hydrogen isotopes Deuterium and Tritium are fusing to helium ions, provided that they can be kept long enough and dense enough at very high temperatures (typically $\gg$ 1 million degrees), thereby delivering energy according to the equation $E = (m_{Deuterium/Tritium} - m_{Helium}) \times c^2$. To control the plasma confinement and to get information about its physical behavior, a high-speed and high-volume data acquisition system is needed for the on-line and off-line monitoring and evaluation of measured plasma and fusion device data. State-of-the-art experiments produce approx. 100 MByte of measurement values during a 10 second experimental period. Future plasma devices will deliver one to two orders of magnitude more data in the same time interval, while additionally operating continuously in a steady-state mode. This imposes high requirements on the real-time behavior, i.e., the transmission latency, as well as on the bandwidth of the underlying communication network that is part of the data acquisition system. A low and guaranteed latency is very important for the closed-loop feedback systems of the fusion device that keep the burning plasma hot, stable, and away from physical material.

Potential candidates for the communication network of such a future data acquisition system are Fiber Distributed Data Interface (FDDI), Gigabit Ethernet, Asynchronous Transfer Mode (ATM), and Scalable Coherent Interface (SCI). Among them, SCI seems to be especially attractive because of its forward progress guarantee, prioritized bandwidth allocation and extremely low latency. For the same reasons, also other high-end physical experiments such as the "Large Hadron Collider" at CERN are considering SCI as the primary communication medium [3, 4, 5, 20, 30]. In industry, SCI gains more and more importance since large computer manufacturers are offering off-the-shelf SCI-based products for cluster computing [22, 7]. However, for data

† Partially reprinted from: H. Richter and M. Ohlenroth: Data acquisition with the SCINET, a scalable-coherent-interface network, *Fusion Engineering and Design*, vol. 43, pp. 393–400, Copyright 1999, with permission from Elsevier Science

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 125-149, 1999
© Springer-Verlag Berlin Heidelberg 1999

acquisition purposes commercially available SCI products are very rare, and for high-end applications no off-the-shelf solutions are provided since this market segment is too small. A lot of research remains to be done in this field [23, 25, 31, 21, 19].

One research project is SCINET, the goal of which is to investigate the applicability of SCI networks for data acquisition systems in large-scale fusion reactor experiments. SCINET investigates how data acquisition computers can be efficiently connected to each other and to sensors and actuators by means of SCI, which topological structures large-scale networks should have, and the bandwidth and latency that can be expected. Therefore, this chapter first presents the results of SCI test beds that were established as a sample SCI-based data acquisition system. The test beds demonstrate up to 45 MByte/s of throughput and $< 5\mu$s latency for end-to-end data transfers. Second, it is shown how commercial SCI switches can be used more efficiently resulting in more than sevenfold higher throughput and half the latency at the same costs. Third, SCI-based Banyan topologies are proposed which are highly efficient for multi-port switches in parallel computers, clusters of workstations, and local area multiprocessors. The networks have up to four times the performance in terms of throughput and latency, as compared to a conventional SCI-based multistage network, while requiring only one fourth of its costs. The prerequisite for the improvements is that some data locality is present in the traffic patterns between senders and receivers. All results were achieved by means of our SCINET simulator.

The chapter is organized as follows. In Section 6.2, the basic requirements of a data acquisition system for a fusion reactor experiment are explained. In Section 6.3, the two SCI test beds used for benchmarking are described. Section 6.4 shows the performance results of the test beds which indicate the principal suitability of SCI for data acquisition systems in plasma physics. Section 6.5 is devoted to SCI switches as the basic components of large-scale data acquisition systems. Section 6.6 explains and validates (by means of simulation) that SCI switches can be used more efficiently. Section 6.7 deals with multistage networks; two new cost-efficient Banyan topologies are proposed that can replace conventional solutions. In Section 6.8, simulation results for the new topologies are given. The chapter concludes with a summary in Section 6.9.

6.2 SCI-based Data Acquisition Systems

Data acquisition is a real-time task with reaction times determined by the sampling rates of the probing devices that collect the measurement values. Each data acquisition system can be evaluated by six key parameters:

- The probes' data rates in terms of samples per second. This value determines mainly the technology that has to be employed. For a fusion reactor,

data normally need not to be sampled with frequencies higher than 100 MSamples/s per sensor.

– The total number of sensors that have to be read out, maintained, and operated over a longer period of time (normally 1-2 decades).
– The amount of data collected by the system. For today's data acquisition systems, this value lies in the range of 10-100 MByte/s. Future systems will deliver 1-2 orders of magnitude more data which imposes high requirements on the database and data mining systems.
– The system's delay time between sampling and remote storing of the measurement values. This is a crucial factor if any additional feed-forward or feedback control system has to make use of the measured values because it determines the time constant, i.e., the reaction delay of the control system.
– The allowable data loss rate of the system. This rate determines the degree of redundancy that has to be built into the system. Since fusion reactor experiments are mission-critical, high reliability is required. Thus, the data loss rate has to be kept significantly lower than in telecommunication systems, for instance.
– The scalability of the system. Large-scale experiments take about 10 years to be built and are operated another 10-15 years. This means that their data acquisition system has to be flexible enough to be expanded over the years.

In this chapter, throughput and latency of a sample SCI-based data acquisition system are investigated. Data losses are partly evaluated by registering the error rates that occur on the transmission lines. Other sources of data loss, for instance those due to buffer overflows, are not considered. The management of the data retrieval and the overall system's scalability are outside the scope of this contribution.

SCI is believed to be a proper technology for high-end data acquisition systems since it has, by construction, guaranteed data delivery, high throughput, low latency and scalability. In the framework of the SCINET project, a sample acquisition system was implemented to validate this. Throughput and latency of data transfers from memory to memory were measured by means of test beds.

6.3 SCINET Test Beds

Two different SCI test beds were set up to enable comparisons of, and to achieve a higher confidence in, the obtained results. Each system represents an SCI data transmission ring, comprising two PCs that are connected by SCI interfaces. For both test systems, commercially available SCI cards from Dolphin Interconnect Solutions [11] are used, together with appropriate copper cabling for the transmission lines.

In the first system, two 200 MHz Pentium Pro PCs with the 440FX PCI chip set are employed (GA- 686DX mother board), running the Linux 2.0.30 operating system. The Dolphin boards are controlled by our own device driver, and also the benchmarks that measure the throughput and latency are self-developed. The second system comprises two 100 MHz Pentium PCs, Windows NT 4.0, and the standard FX chip set. Here, device driver and benchmark software were delivered by Dolphin. In both cases, the responseless DMOVE64 transaction of SCI was used for all data transfers.

Each interface card is based on Dolphin's 200 MByte/s Link Controller (LC) chip [10, 12] that implements the physical layer of SCI, and a PCI bridge that converts the PC's PCI bus protocol into SCI packets. On the card, the PCI bridge and the LC chip are connected internally via a proprietary high-performance bus, the so called B-Link [8]. The LC conforms to the IEEE interface specification, with the exception that the attached user device is a PC that is connected via the PCI/B-Link bridge to the SCI interface. The bridge consists mainly of a PCI master/slave building block, a DMA engine and 8 buffers for read and write transactions, respectively. Further components are an address translation cache for the mapping of PCI to SCI addresses, configuration, control, and status registers, and an interrupt mechanism. The block diagram of the PCI/B-Link bridge is shown in Figure 6.1. The adapter card is further described in Chapter 3.

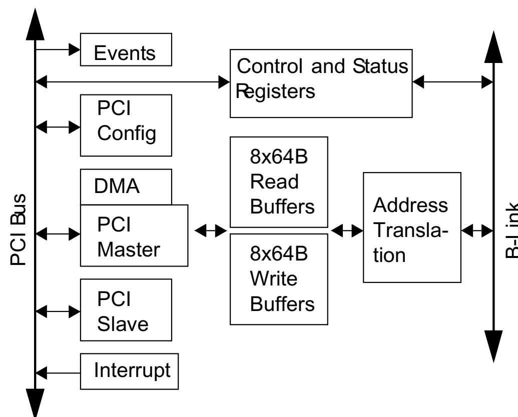


Fig. 6.1. Dolphin's PCI/B-Link bridge

The bridge is capable of either splitting its buffers in order to support up to 8 external PCI masters, for read and write, respectively, or of combining the buffers to allow for eightfold throughput for a single master. The prerequisite for buffer combining is that PCI data that has to be transferred to the SCI link

is located on contiguous addresses, and that the control and status registers of the bridge are properly set.

6.4 Measurement Results

The throughput results shown in Figure 6.2 were obtained for both, remote read and write transactions between the memories of the Pentium Pro test bed (system 1). Data transfer took place exclusively between user address spaces, thus the values represent the real end-to-end data rate. As one can see, the SCI ring already reaches its maximum throughput at short block lengths of 256 bytes; at 64 bytes, about half of that can be achieved. Remote read is significantly slower than remote write, by a factor of 3.5. This stems from the fact that, for the requester, a remote write transaction is already finished as soon as its data are stored in the transmit buffer of the PCI/B-Link bridge. From there on, reliable delivery is guaranteed. In contrast, a remote read always needs to wait until the requested data is obtained because of the semantics of the read transaction. Obviously, to achieve maximum throughput only remote writes should be employed. This means for the data acquisition system that a "push" strategy should be preferred to "pull"-style operation. That is, the sensors should on their own write their measured data to the final destinations after they have received a data-sampling trigger from those locations, and the remote computers should not read out the probes. For that reason, only remote write transactions are further investigated in the following.

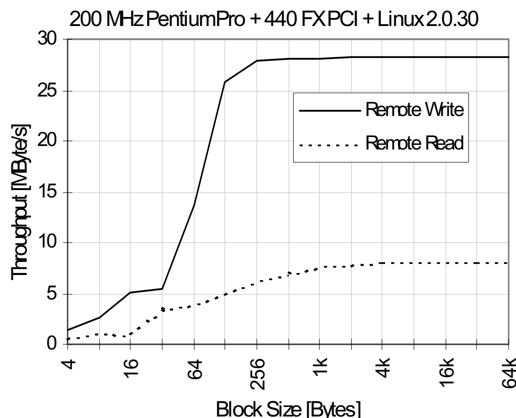


Fig. 6.2. Throughput on system 1

When one considers the elapsed times of the data transfers vs. the block size (Figure 6.3), it becomes clear that the setup times to initiate a transfer

are quite small: < 10μs for both, read and write. That would allow a very small reaction time of an SCI-based control system. The elapsed times for remote read (RR) are higher, compared to remote write (RW) due to its slower transfer rate.

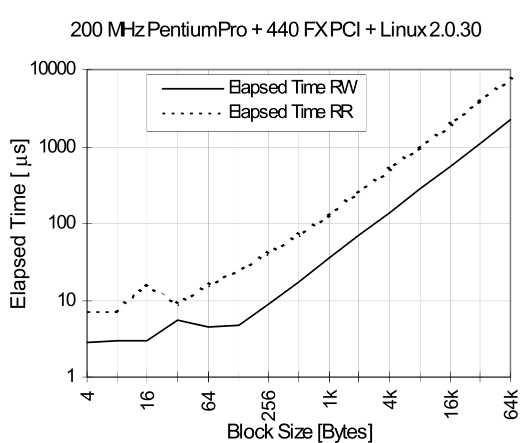


Fig. 6.3. Latencies of remote read and write on system 1

The Pentium-based test bed with the Dolphin software (system 2) shows a throughput behavior as depicted in Figure 6.4 for comparison; here, only 12 MByte/s can be achieved. Additionally, Figure 6.4 shows how the transfer rate behaves in case that, on top of SCI's hardware error detection and correction, a software error check with optional retry of the last sent block is applied. For both cases (HW correction only and HW+SW correction), the maximum throughput turns out to be the same, but for the latter it can be achieved only for large block sizes of > 64 kByte. Without software overhead, Dolphin's solution reaches its maximum transfer rate at 64 bytes already. The latencies for larger block sizes (depicted in Figure 6.5) are in compliance with the measured throughput. For small blocks, the latency becomes as low as 2-3 μs. Of course, more time is needed (11-12 μs) if additional software error correction is used.

Dolphin's device driver allows the combining of the PCI/B-Link bridge write buffers. As one can see from Figure 6.6 and Figure 6.7, the throughput scales nearly linearly with 2 and 4 buffers combined. Only the minimum required buffer size for full transmission speed is doubled each time.

The maximum achievable data transfer rate is 45 MByte/s since with 8 combined buffers saturation effects are showing up (Figure 6.8). Since each write buffer has a capacity of 64 bytes, the combining of 2 or 4 buffers requires that at least 128 bytes or 256 bytes, respectively, that are aligned to 64-byte

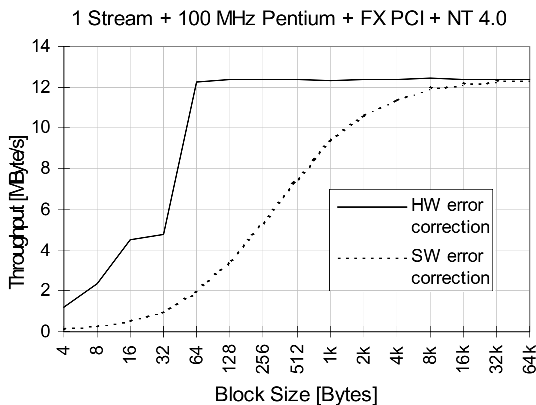


Fig. 6.4. Throughput of 1-stream remote write on system 2

.

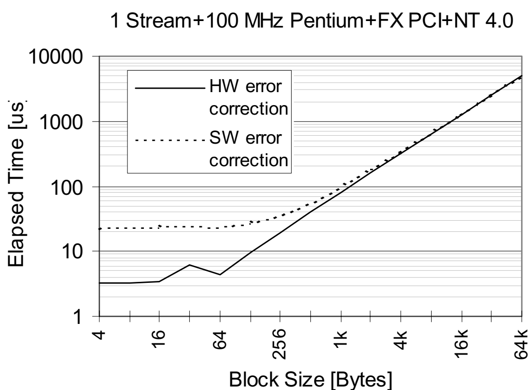


Fig. 6.5. Latencies of 1-stream remote write on system 2

.

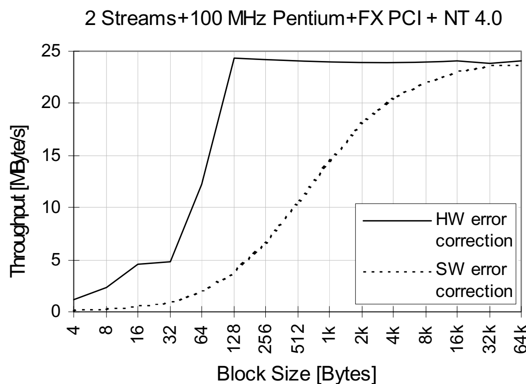


Fig. 6.6. Throughput of 2-streams remote write on system 2

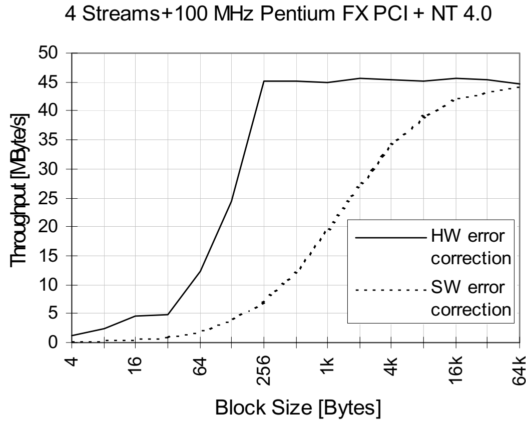


Fig. 6.7. Throughput of 4-streams remote write on system 2

boundaries in the PCI address space, are available for transfer. At the latter block size, maximum throughput is achieved. In the case of combining 8 buffers, bursts of 512 bytes and larger suffer to be transferred at a reduced speed of only 34 MByte/s. Either the PCI bus, the slow Pentium CPU or some other limiting factor induced this saturation effect that causes the significant performance drop.

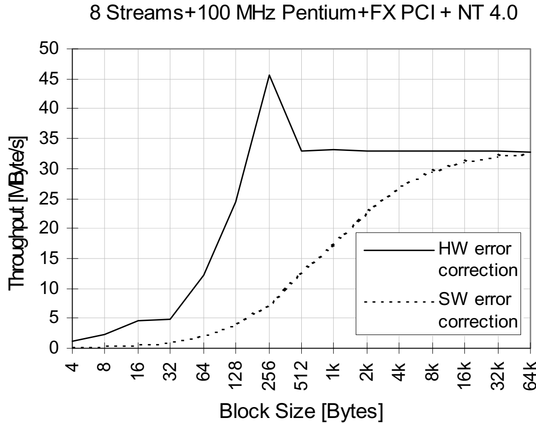


Fig. 6.8. Throughput of 8-streams remote write on system 2

The latency times for 2, 4, and 8 combined streams are given in Figures 6.9, 6.10, and 6.11, respectively.

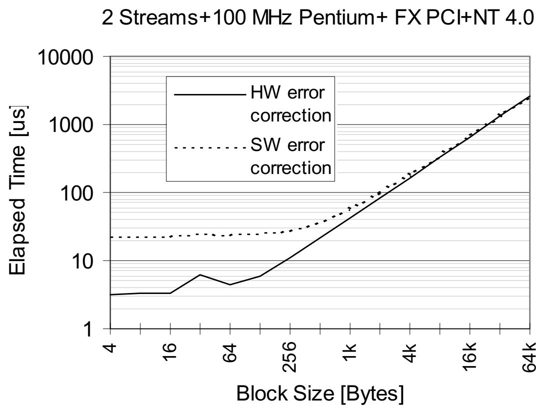


Fig. 6.9. Latency of 2-streams remote write on system 2

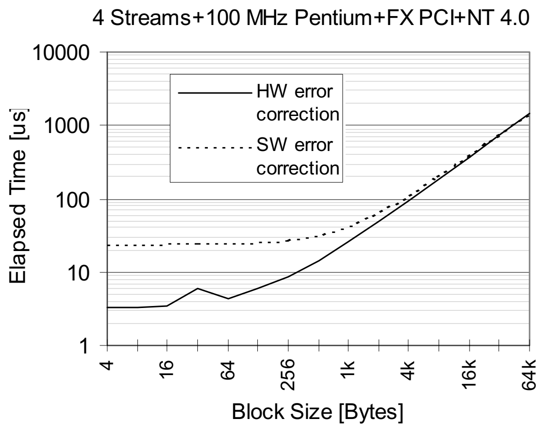


Fig. 6.10. Latency of 4-streams remote write on system 2

.

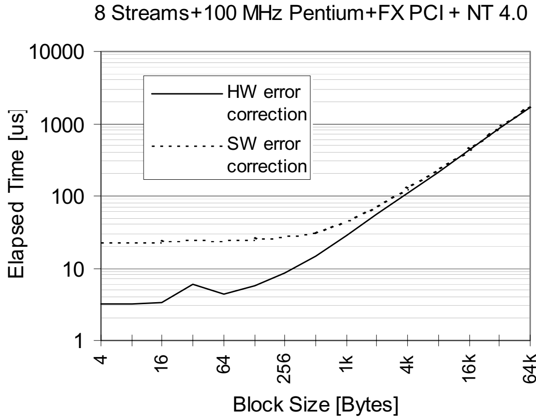


Fig. 6.11. Latency of 8-streams remote write on system 2

6.5 SCI Switches

In addition to commercial SCI products for data transfer between PCs and workstations, SCI switches [9] were investigated, since they are the prerequisite for large-scale data acquisition systems. Switches allow to connect two or more SCI rings, thereby forming a static or dynamic SCI network. Each SCI network can be composed of nodes such as computers, processors, memories, peripherals, routers, bridges, and switches. By proper address management, a commercial 4-port SCI switch can act as a router, if it is connected with one port to an SCI node and with the remaining ports to a static network such as a torus. It can act as a bridge if it is located between adjacent rings to allow data to pass, and it can be employed to establish multistage networks (Figure 6.12). An SCI switch differs in various respects from conventional switches. First, each SCI switch is part of 2 to 4 rings on which data are unidirectionally transferred. Second, there exists a port-internal bypass FIFO connecting the in and out terminals of each port to allow a very fast bypass (< 50 ns). Third, in each port two separate buffers for SCI requests and responses are available preventing deadlocks caused by cyclic waiting on resources (Figure 6.13).

In the following, we consider switches according to Dolphin's implementation [9]. The transmit and receive buffers of the ports of such a switch are connected to a high-speed packet bus called B-Link [8], which has a transmission rate of 600 MByte/s. Inside the switch, the B-Link connects 4 ports, each of which has a data rate of 500 MByte/s per direction. By this, a high-speed SCI switch is established. Between any pair of ports, the maximum port rate of 500 MByte/s can be achieved for unidirectional transfers (either read or write) as long as the remaining other pair of ports produces no more than 100 MByte/s of traffic. If both pairs simultaneously operate in full duplex

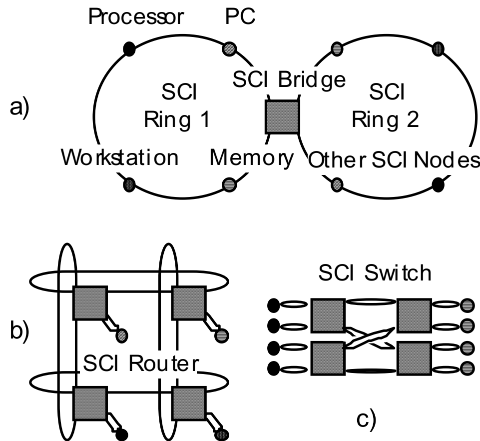


Fig. 6.12. SCI switches can serve as bridges (a), routers (b), and building blocks for multistage networks (c)

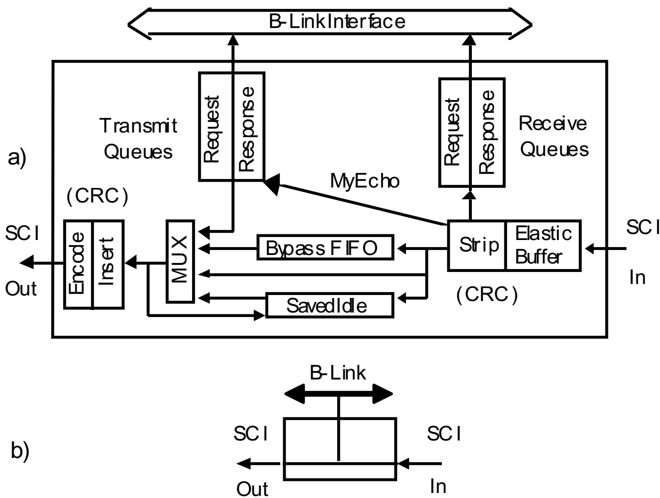


Fig. 6.13. SCI switch port (a) and its symbolic representation (b)

mode, the individual port rate per direction is reduced to 150 MByte/s due to the B-Link's bandwidth limitations.

For illustration, Figure 6.14 depicts a block diagram of a 4-port SCI switch as well as its equivalent representation that will be used later in this chapter.

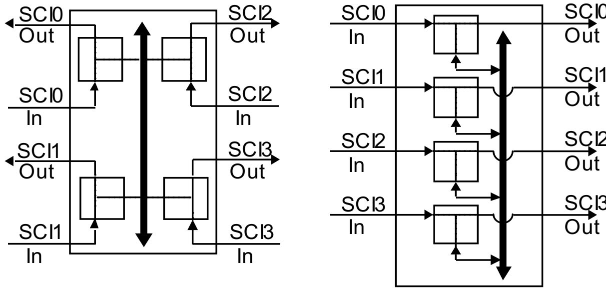


Fig. 6.14. Two equivalent representations of a 4-port SCI switch

6.6 Efficient Use of SCI Switches

Let the throughput T of a switch be the sum of the ports' throughputs. For a subsequent comparison with SCI systems, Figure 6.15 shows two simple multiprocessors (UMA and NUMA variants) that employ conventional switches.

In Figure 6.15(a), the throughput of the conventional switch is assumed to be T_{con}, with $T_{con} \leq 2t$, where t is the throughput of a single switch port to which a processor is connected. In the NUMA example of Figure 6.15(b), every pair of computing nodes can simultaneously communicate with each other (up to two pairs at the same time), thus pushing the throughput to T'_{con} with $T'_{con} \leq 4t$. In both cases, the switch-internal transfer capacity B_{tr} is assumed to be sufficiently large to carry the produced traffic ($B_{tr} \geq T'_{con} \geq T_{con}$).

With SCI, it is possible to push T'_{con} above the bandwidth limit B_{tr} by using the ports' bypass FIFOs for additional data transfers. This special switch usage will be explained in the following. In Figure 6.16(a), the UMA architecture of Figure 6.15(a) is upgraded to an SCI switch, and the bidirectional transmission lines are replaced by SCI ringlets. Now, the total throughput is T_{SCI}, with $T_{SCI} = min\{2t,\ B_{tr}\}$ which is the same as with conventional switches. This solution is published in the literature [20, 30]. In Figure 6.16(b) however, the processor and memory nodes are coupled differently: the SCI ringlets are replaced by long rings connecting a sender, a switch, and a receiver in one instead of two rings so that no B-Link is in between.

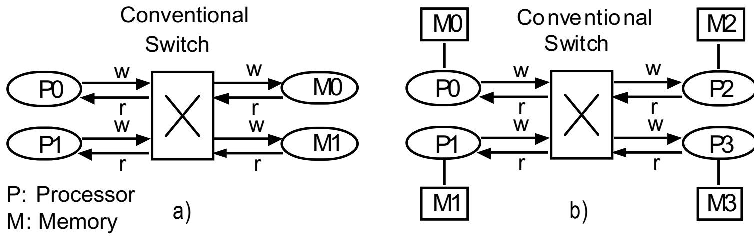


Fig. 6.15. Simple UMA (a) and NUMA (b) multiprocessors based on conventional switches

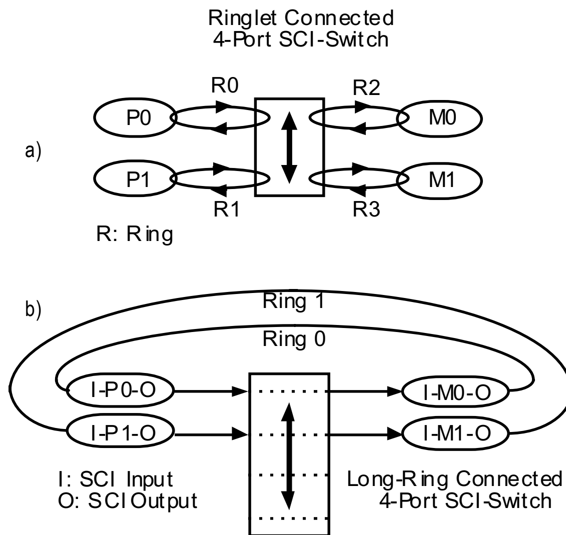


Fig. 6.16. SCI switch using B-Link (a) or bypass FIFOs (b) as main data paths

Here, we obtain throughput T'_{SCI} which can become larger than B_{tr}, provided that some fraction of the data can stay on the ring where it originated. The reason for higher throughput is that data may enter and leave the switch through the ports' bypass FIFOs, so that the B-Link bottleneck is circumvented.

The prerequisite that T'_{SCI} exceeds B_{tr} is that the communication patterns between senders and receivers exhibit some data locality. Data locality is common to most parallel applications; if not, it can be explicitly forced by proper allocation of tasks and data structures to computing nodes. In the example depicted in Figure 6.17(a), this means that processor $P_i (i = 0, 1, 2, 3)$ mainly communicates with memory M_i, so that data packets can stay on the rings where they originated and can travel to the destinations through the bypass FIFOs.

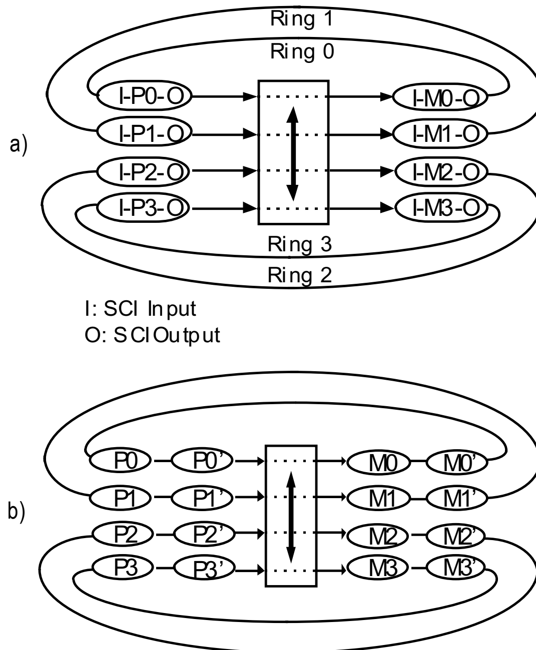


Fig. 6.17. Flexibility in the number of nodes by means of long rings

In addition, the latency is reduced since in SCI the intra-ring communication is faster than the inter-ring communication. Furthermore, from Figure 6.16(a) to Figure 6.16(b) the amount of required hardware has decreased from 4 to 2 rings and from a 4-port to a 2-port switch while the performance

is expected to increase. For larger switch sizes than 4 the same improvement would be achieved.

Finally, as shown in Figure 6.17(a) and (b), more flexibility with respect to the number of attachable processors and memories can be obtained by using long rings that pass through SCI nodes and switches. In the example of Figure 6.17(a), twice as many processors and memories are coupled to the same 4-port switch without degradation in performance, as compared to the ringlet configuration. In the example of Figure 6.17(b), the number of connectable devices is again doubled. However, in the latter case, the maximum data rate per processor may be halved. In Section 6.8, the predicted performance improvements are quantified.

6.7 Multistage SCI Networks

In this section, the long-ring connection technique is applied to multistage networks comprising 2-port or 4-port SCI switches, in order to construct more efficient networks. Generally, the most cost-efficient multistage networks are of the Banyan type [15], since they can be built with the minimum number of stages. However, Banyans are blocking networks which do not have redundancy and therefore also no fault tolerance. Typical Banyans are Baseline, Omega, Flip, Butterfly, Indirect Binary n-Cube, and Generalized Cube networks [28].

In Figure 6.18(a), the standard implementation of an SCI-based Baseline network according to [30] is shown: nodes and switch ports of adjacent stages are connected by SCI ringlets. A functional equivalent but bypass FIFO-based solution that consists of large toroidal rings is shown in Figure 6.18(b).

In this example, the switch complexity is reduced from 4-port to 2-port switches. The minimum latency of data to travel from an input to an output of a network of size N has decreased from $L = \alpha \log_2 N$ to $L' = \beta \log_2 N$. The factor α denotes the time for a packet to cross a switch (usually some μs), while β is the time to travel through a bypass FIFO (some tens of ns). Obviously, two orders of magnitude in latency decrease can be expected by employing long rings instead of ringlets, while halving the switch costs at the same time.

A disadvantage of the Baseline network of Figure 6.18(b) is that an additional permutation wiring is required to connect the memory-out links *a*–*h* with the corresponding processor-in links to obtain closed rings. Fortunately, by virtue of their topological structure, two of the known Banyans allow a one-to-one connection from outputs to inputs. These topologies are the Omega and the Generalized Cube networks. (Their “mirror images”, the Flip network and the Indirect Binary n-Cube, have the same property.) Therefore, we propose that SCI networks which have bypass FIFOs as their main data paths should be built according to one of these 4 topologies. In the following,

.

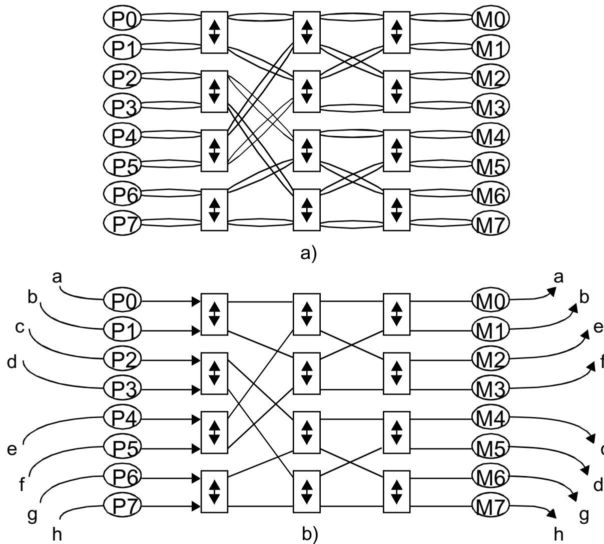


Fig. 6.18. Baseline networks with B-Links (a) or bypass FIFOs (b) as main paths

the networks without permuted wiring from output to input are termed *first-grade optimized*. An example of such a network is given in Figure 6.19.

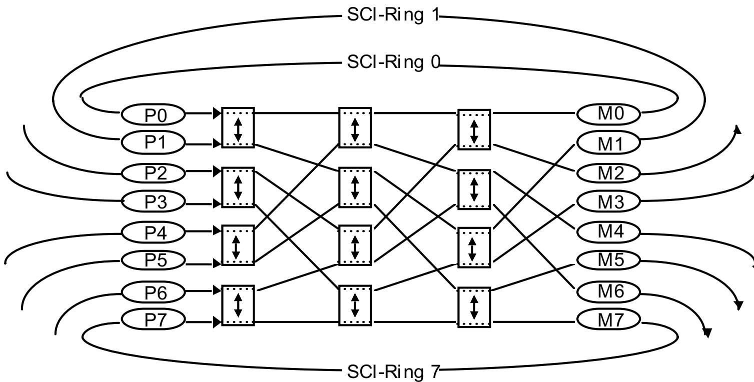


Fig. 6.19. First-grade optimized SCI-based Omega network

First-grade optimized SCI Banyans can be further improved by using s-port switches with $s > 2$, which results in additional improvements by a factor of $\log_2 N/log_sN$ in terms of costs and latency. For example, when $s = 4$ and a network size of 16×16 is assumed, then 32 ports are necessary to build up all network switches, while the equivalent first-grade optimized network

with $s = 2$ needs twice as many, i.e. 64 ports. Since the port count is the dominant cost factor of a network, the prize is approx. halved for $s = 4$. A ringlet network of the same size and $s = 2$ would require 128 ports.

If $s > 2$, we call such a structure a *second-grade optimized* network. An example of a second-grade optimized network is depicted in Figure 6.20. In Section 6.8, the performance of first-grade and second-grade optimized networks will be compared.

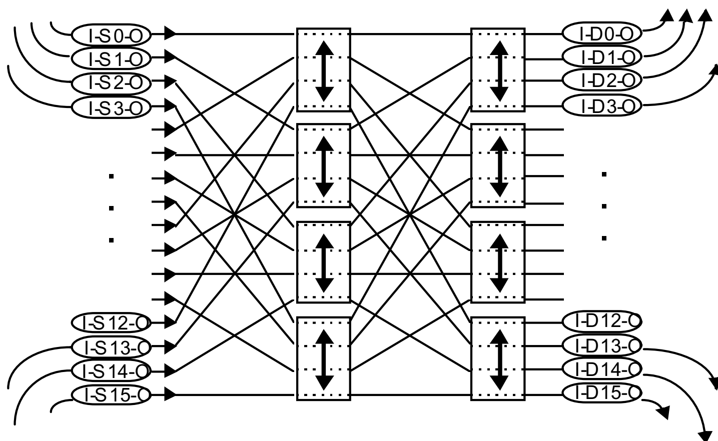


Fig. 6.20. Second-grade optimized SCI-based Omega network

6.8 Simulation Results

A first suite of simulations was conducted to evaluate the performance of a single 4-port SCI switch, which uses ringlets to connect processors and memories, and to compare that solution with a switch based on long rings (Figure 6.16). The performance metrics are throughput, latency, and packet losses. To pinpoint the performance discrepancy between both concepts, 100% data locality was chosen, i.e. processor P_i communicates exclusively with memory $M_i, i = 0, 1$. In practice, the locality will be lower, but as long as there is some data locality, a performance improvement will be visible.

For all simulations, the DMOVE64 SCI command was chosen, and all processors are configured to simultaneously send DMOVE64 packets at the same rate. The input data rate to the switch was decided to be deterministic, no random traffic is applied.

In the following graphs, the achieved data throughput of the switch (net output rate) versus the generated input traffic (gross input rate) are shown.

The input is varied from 0 to 500 MByte/s per processor, which is also the maximum ring speed, to study the input/output behavior of the switch. The data packets carry 64 bytes of payload with an overhead of 16 bytes for header and trailer. Together with additional 4 bytes for idle symbols, the ratio of payload length to raw length is 64/84. The memories are assumed to have an access time of 40 ns for each block of 64 bytes. The link delays between processors, switch, and memories are set to be 1 ns each. The remaining timing parameters for all SCI ports are 20 ns address decoder delay, 48 ns bypass FIFO delay, 106 ns FIFO to B-Link delay and 82 ns B-Link to FIFO delay. All ports are modeled to have input and output buffer space for 4 request and response packets each. By this parameter set, the simulations are compliant with the latest SCI Link Controller (LC-2) of Dolphin [12].

The achieved throughput rates of a ringlet and a long-ring connected 4-port switch are shown in Figure 6.21. With ringlets, the switch already saturates at 250 MByte/s raw input rate, delivering 176 MByte/s output payload. At the same input rate, the packet losses become significant and eventually reach a value of 585 MByte/s at 1 GByte/s gross input rate. A packet loss occurs each time a new packet is generated that cannot be injected into the ring by a sender's SCI interface; this happens when the ring is still occupied by transferring previous packets and the transmit buffer of the interface is full. Because of the constant rate with which data are issued by the processors, the ring has to accept packets in real time which is only possible up to a certain speed. Above that limit, packets are lost.

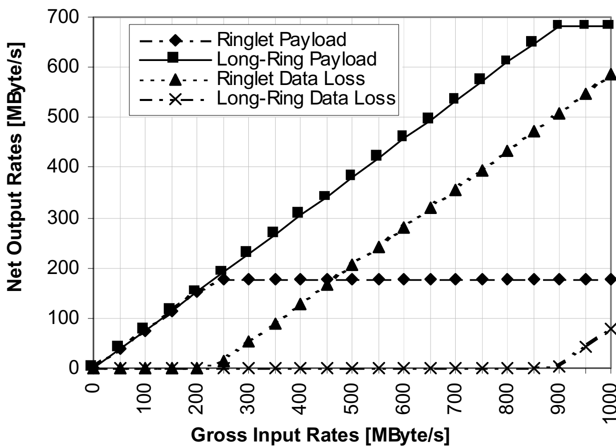


Fig. 6.21. Throughput and packet losses of ringlet and long-ring connected 4-port SCI switches with two senders and receivers

The latency behavior of a ringlet-connected 4-port switch is depicted in Figure 6.22. The time from initiating a DMOVE64 packet to storing it in its destination shows a value of 2344 ns as long as the latency saturation point of 200 MByte/s is not reached. The latency increases and becomes non-deterministic above that point, assuming values between 7362 ns (minimum) and 12797 ns (maximum).

.

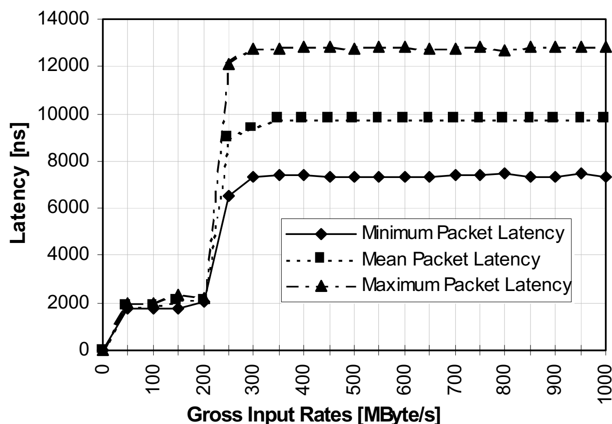


Fig. 6.22. Latency of ringlet-connected 4-port SCI switch with two senders and receivers

The long-ring connected switch coupling two senders and receivers behaves much better: it saturates at 900 MByte/s gross input rate with 682 MByte/s output payload, and at 1 GByte/s input rate it has 80 MByte/s packet losses. Below the saturation point, a latency of 1127 ns can be expected. Compared to the ringlet-case, the throughput has increased by a factor of 3.9 while the latency has decreased by 52%. However, above the saturation point, latency not only becomes non-deterministic but it also jumps by two orders of magnitude, varying between 75 μs and 301 μs (Figure 6.23). Obviously, for 100% data locality the long-ring connected switch behaves much better than a conventionally ringlet-coupled one, but it should not be overloaded.

If all 4 ports of the switch are coupled with long rings to 4 processors and memories, as depicted in Figure 6.17(a), a linear performance improvement compared to 2 processors and memories is achieved. As shown in Figure 6.24, the switch saturates at 1800 MByte/s gross input rate with 1365 MByte/s output payload, and it has 160 MByte/s packet losses at 2 GByte/s. With 1127 ns, the latency below the saturation point is identical to the case of 2 processors and memories. Identical latencies are also obtained above the saturation point.

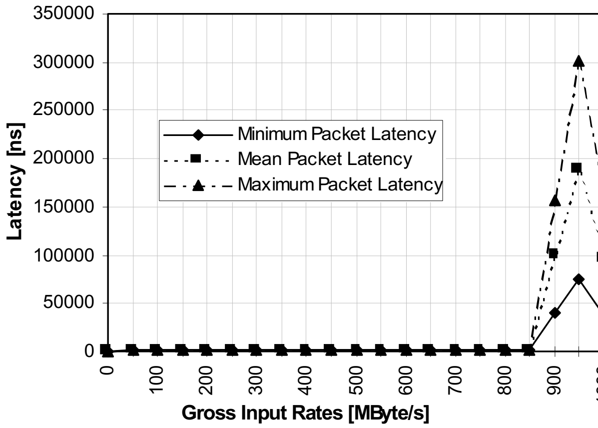


Fig. 6.23. Latency of long-ring connected 4-port SCI switch with two senders and receivers

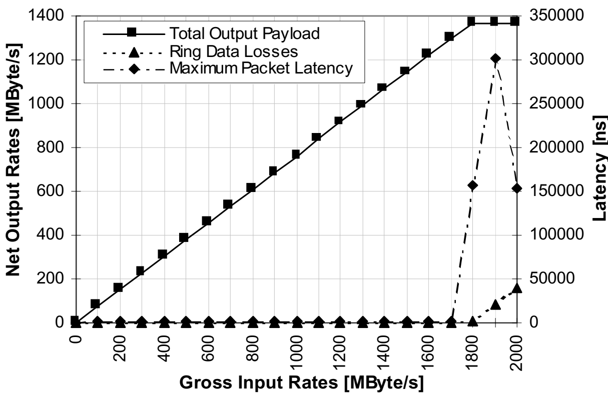


Fig. 6.24. Performance of 4-port switch with 4 processors and memories

This means that with the same switch as in the ringlet-connected case, a 7.8-fold throughput improvement can be achieved, provided that 100% data locality is present. Latency drops to one half if the switch is not overloaded. Other simulations show that ringlet and long-ring switches will be identical in performance if no data locality is present. This means that for operations below the saturation limit the long-ring coupled switch can always be preferred.

The second series of simulation experiments that was performed with the SCINET tool compares a 16×16 first-grade optimized Omega network with a conventional ringlet-based network of the same size and topology. Furthermore, first-grade and second-grade optimized networks are compared. Again, 100% data locality is assumed to demonstrate the upper limit of the performance improvements.

The behavior of the 16×16 ringlet-based SCI network is depicted in Figure 6.25. The network saturates at 2 GByte/s gross input rate with 1412 MByte/s output rate. At that point, the network has 112 MByte/s packet losses that increase up to 4684 MByte/s at 8 GByte/s input rate. The maximum latency is 6670 ns below saturation and jumps up to 20056 ns above saturation. Latency saturation occurs earlier than throughput saturation, at 1600 MByte/s.

.

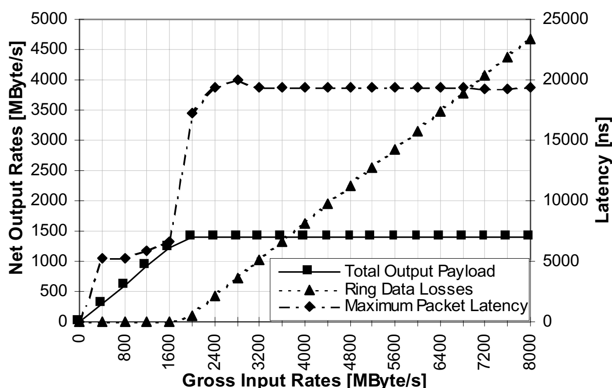


Fig. 6.25. Performance of a ringlet-connected 16×16 Omega network with 100% data locality

The performance of the first-grade optimized Omega network is shown in Figure 6.26. It has 5333 MByte/s output rate at 7200 MByte/s gross input rate, a 3.8-fold performance improvement over the ringlet network. The latency is much better as well: below the latency saturation point of 6800 MByte/s, we have 1771 ns which are 27% of the ringlet latency. Above

that point, a maximum of 5067 ns is reached which is roughly one fourth of the ringlet's latency.

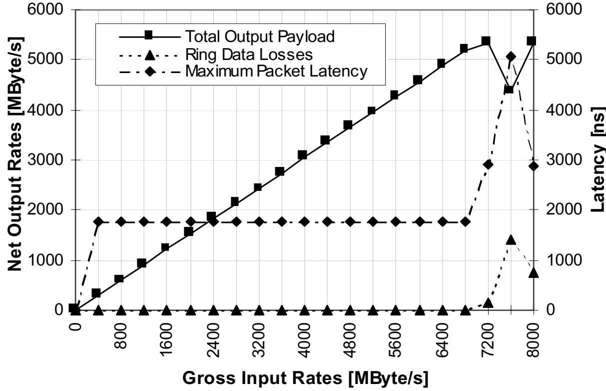


Fig. 6.26. Performance of first-grade optimized 16×16 Omega network with 100% data locality

This means that the first-grade optimized network of size 16×16 has approx. the 4-fold throughput and one fourth of the latency of a conventional SCI-based Omega network, at only half of its costs. The results can be extrapolated to sizes > 16.

The simulation results for the second-grade optimized network are shown in Figure 6.27. It can be seen that throughput saturation also occurs at 7200 MByte/s gross input rate, but with 5456 MByte/s output rate it delivers a slightly higher throughput than the corresponding first-grade optimized network. Also the latency before saturation is better, 1525 ns. However, after saturation latency jumps by two orders of magnitude and reaches 134 μs. Note that the fully connected 4-port switch shows the same behavior.

The second-grade optimized network delivers slightly better performance in all respects compared to the first-grade optimized network as long as it is not overloaded, but at only half the costs. Both have roughly four times the performance in terms of throughput and latency of the ringlet-coupled network of the same type and size, while requiring only one half or one fourth of the costs, respectively.

6.9 Summary and Conclusions

In this chapter, the feasibility of an SCI-based data acquisition system was demonstrated by the achieved throughputs of 45 MByte/s and latencies of

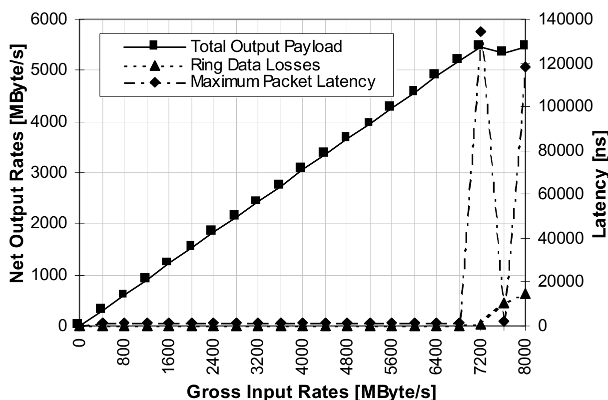


Fig. 6.27. Performance of second-grade optimized 16 × 16 Omega network with 100% data locality

$< 10\mu$s. The sample DAQ system was based on PC test beds where a decent performance was also measured for the case of additional software error checking and correcting. In the future, the test beds will be upgraded to fiber links and an SCI switch will be included to study its influence in practice. Additionally, the interactions of the transmission system and the higher software levels have to be analyzed, and a programming model must be devised that is suitable for the physicists' needs.

Furthermore, we have shown by means of simulation how commercial SCI switches, which are the basic blocks of *large-scale* data acquisition systems can be used more efficiently. The principle is to use the bypass FIFO that is part of every SCI port, for establishing long SCI rings comprising a pair of nodes where one node is a transmitter and the other node is a receiver. Then, packets can be redirected from the switch-internal bus (i.e. the B-Link) which is a bottleneck, to the port's bypass FIFO. The prerequisite for redirection is that there exists data locality in the traffic pattern between the sender and the receiver residing on the same ring. For 100% data locality, up to a 7.8-fold throughput improvement is achievable compared to a ringlet-connected switch. Latency is reduced by a factor of 2 and the number of lost packets by a factor of 7. Additionally, by grouping pairs of sender and receiver nodes into each ring, more nodes can be connected. However, latency increases by two orders of magnitude above the saturation point of the ring.

A new network type, called first-grade optimized network, was proposed. It is based on long rings passing through all switch stages of a Banyan network. Each long ring replaces a number of ringlets connecting neighboring nodes in the conventional network. The new network type improves multistage Banyan networks that are based on SCI ringlets. With long rings and data locality, packets can stay on the ring where they originated, thus remo-

ving traffic from each switch-internal bus (B-Link). For 100% data locality, a first-grade optimized network has four times the performance in terms of throughput and latency of a conventional SCI network of same size and type while exhibiting one half of its costs.

Finally, so-called second-grade optimized networks were suggested, further improving first-grade ones. They use long rings as well as intra-stage wirings with permutation functions on a number base higher than two. With a permutation base of four, for instance, one half of the costs of a first-grade optimized network can be achieved, even with slightly better performance in throughput and latency. The performance improvement is proportional to the number of network ports. All results were obtained by the newly developed SCINET simulation program.

References

1. T. E. Anderson, D. E. Culler, D. A. Patterson, A Case for NOW (Networks of Workstations). *IEEE Micro*, pages 54–64, Feb. 1995.
2. N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb. 1995.
3. A. Bogaerts, R. Divia, H. Müller, J. Renardy. SCI-based Data Acquisition Architectures. *IEEE Transactions on Nuclear Science*, Vol. 39, No. 2, Apr. 1992.
4. A. Bogaerts, R. Keyser, G. Mugnai, H. Müller, P. Werner, B. Wu, B. Skaali, J. Ferrer-Prietro. SCI Data Acquisition Systems: Doing More with Less. *CHEP'94*, San Francisco, April 1994
5. A. Bogaerts et al. *RD 24 Status Report: Application of the Scalable Coherent Interface to Data Acquisition at LHC*. Oct. 1996. http://nicewww.cern.ch/~hmuller/~HMULLER/docs/report96.pdf.
6. CACI Products Company. *Modsim II, The Language for Object Oriented Programming*. Reference Manual, La Jolla, California, 1995.
7. R. Clark, K. Alnes. An SCI Interconnect Chipset and Adapter. *Proc. Hot Interconnects Symposium IV*, Stanford University, Aug. 15-17, 1996.
8. Dolphin Interconnect Solutions. *A Backside Link (B-Link) for Scalable Coherent Interface (SCI) Nodes*. Dolphin Interconnect Solutions Inc., Oslo, Norway, 1994.
9. Dolphin Interconnect Solutions. *4-way SCI Cluster Switch*. Dolphin Interconnect Solutions Inc., Oslo, Norway, 1995.
10. Dolphin Interconnect Solutions. *Link Controller LC-1 Specification*. Dolphin Interconnect Solutions Inc., Oslo, Norway, 1995.
11. Dolphin Interconnect Solutions. *PCI/SCI Cluster Adapter Specification*. Dolphin Interconnect Solutions Inc., Oslo, Norway, 1996.
12. Dolphin Interconnect Solutions. *Link Controller LC-2 Specification*. Dolphin Interconnect Solutions Inc., Oslo, Norway, 1997.
13. D. R. Engebretsen, D. M. Kuchta, R. C. Booth, J. D. Crow, W. G. Nation. Parallel Fiber-Optic SCI Links. *IEEE Micro*, pages 20–26, Feb. 1996.
14. R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, pages 12–19, Feb. 1996.
15. L. R. Goke, G. J. Lipovski. Banyan Networks for Partitioning Multiprocessor Systems. *Proc. 1st Int'l. Symposium on Computer Architecture*, pages 21–28, 1973.

16. D. B. Gustavson, Q. Li. Local Area Multiprocessor: the Scalable Coherent Interface. *Defining the Global Information Infrastructure*, S. F. Lundstrom (ed.), SPIE Press, Vol. 56, pp. 141–160, 1994.
17. Standard for Scalable Coherent Interface (SCI). *IEEE Std. 1596-1992*
18. Standard for Heterogeneous Interconnect (HIC). *IEEE P1355 Proposed Standard*
19. D. V. James. The Scalable Coherent Interface: Scaling to High-Performance Systems. *Proc. COMPCON Spring'94*, 1994.
20. E. H. Kristiansen, G. Horn, S. Linge. Switches for Point-to-Point Links Using OMI/HIC Technology. *Int. Data Acquisition Conference on Event Building and Data Readout*, Fermi National Accelerator Laboratory, Batavia, Illinois, USA, Oct. 1994.
21. M. Liebhart, A. Bogaerts, E. Brenner. A Study of an SCI Switch Fabric. *Proceedings IEEE MASCOTS'97*, Haifa, Israel, 1997.
22. K. Omang, B. Parady. Performance of Low-Cost UltraSparc Multiprocessors Connected by SCI. *Proceedings Communication Networks and Distributed Systems Modeling and Simulation (CNDS'97)*, Phoenix, Arizona, USA, Jan. 1997.
23. H. Richter, M. Liebhart. Performance Optimizations of Switched SCI-Rings. *Proceedings 11th Annual International Symposium on High Performance Computing Systems (HPCS'97)*, Winnipeg, Canada, July 1997.
24. H. Richter. *Interconnection Networks for Parallel and Distributed Systems* (in German). Spektrum Akademischer Verlag, Heidelberg, Germany, 1997.
25. S. Scott, J. Goodman, M. Vernon. Performance of the SCI Ring. *Proc. 19th Int'l. Symp. on Computer Architecture*. ACM Press 1992.
26. S. Scott. The GigaRing Channel. *IEEE Micro*, pages 27–34, Feb. 1996.
27. H. J. Siegel, S. D. Smith. A Study of Multistage SIMD Interconnection Networks. *Proc. 5th Int'l. Symposium on Computer Architecture*, pages 9–17, April 1978.
28. C. I. Wu, T. Y. Feng. On a Class of Multistage Interconnection Networks. *IEEE Transactions on Computers*, Vol. C-29, No. 8, pages 694–702, August 1980.
29. B. Wu. Applications of the Scalable Coherent Interface in Multistage Networks. *IEEE TENCON*, Aug. 1994.
30. B. Wu. SCI Switches. *Int'l. Data Acquisition Conference on Event Building and Data Readout*, Fermi National Accelerator Laboratory, Illinois, USA, Oct. 1994.
31. B. Wu, A. Bogaerts, B. Skaali. A Study of Switch Models for the Scalable Coherent Interface. *Proceedings of the Sixth IFIP WG6.3 Conference on Performance of Computer Networks*, Istanbul, 1995.

7. Scalability of SCI Ringlets

Geir Horn

SINTEF Electronics and Cybernetics & University of Oslo,
Forskningsveien 1, P.O. Box 124 Blindern,
N-0314 Oslo, Norway
email: Geir.Horn@ecy.sintef.no
http://www.sintef.no/

7.1 Do SCI Ringlets Scale in Number of Nodes?

SCI[1] originated as a bus replacement technology, where the nodes were envisioned to be board level components like processors, memory, and input and output units. The number of nodes on a ringlet was assumed to be small, hence the adaptation of the terminology *ringlet* as opposed to a presumably larger *ring*. However, SCI has since long made its way into the user area as a *System Area Network*, used for workstation clustering, high performance I/O networks, and high bandwidth data acquisition systems. The scalability in performance for these applications is often a question of the number of nodes that can be put on the ringlet.

Scalability in this sense is a question of bandwidth available to the individual nodes on the ringlet. Contrary to buses, the SCI links will always have the same capacity independently of the number of nodes, however with more nodes on the ringlet a higher fraction of a node's output link will be consumed by bypassing traffic. Due to SCI's concept of request-response traffic, the concept of locality is less clear on the ringlet: if a node a sends a request to node b, the response from node b has to go around the ringlet back to a. As an intuitive approach, for a moment ignorant of the difference in packet sizes between the different transaction types, this request-response transaction can be thought of as a single packet that has to go all the way around the ringlet. Thus each node consumes not only bandwidth from its own output link, but equally much from all the other nodes on the ringlet. Consequently one may expect that with n nodes on the ringlet, the bandwidth available to the individual nodes will be of order $O(1/n)$; and the number of nodes that can be put on a ringlet depends on how little bandwidth each node can accept.

The term *Scalable* in SCI refers mainly to the cache coherence protocol. There are numerous studies like [4] and [6] of this scalability suggesting extensions improving it. However, there has been little interest in bandwidth scalability. Some early studies dealt with this issue through simulation, but then only for systems disabling the request-response transactions through

[1] See [2] for a good first introduction or the standard [3] for the details.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 151-166, 1999
© Springer-Verlag Berlin Heidelberg 1999

using only `move` packets: examples are [5] and [1], or [28] for data acquisition applications within high energy physics. It is only recently with the advent of increasingly performant PCs and workstations that the scalability of the ringlet bandwidth has become an issue [11, 8, 7].

To evaluate the scalability of an SCI ringlet this chapter will present a mathematical model for the bandwidth of an SCI ringlet based on the model in [10], which is the only known previous model for SCI ringlets. The enhanced model is presented in Section 7.2 and used in Section 7.3 to evaluate the maximum number of nodes an SCI ringlet can support given the per node bandwidth requirement. The limitations of the suggested approach and some guidelines on how to circumvent some of these are presented in Section 7.4.

7.2 Ringlet Bandwidth Model

7.2.1 Transaction Formats

The SCI standard specifies a set of different packet types as given in Table 7.1. In general a transaction is either of remote write or remote read nature. For a remote read the requesting node asks for some data stored at some remote memory location and gets a response back containing the data actually stored at that location. For a write transaction, one node sends off the data that should be written remotely, and then gets a confirmation back from the remote node that the data have been successfully written to the remote physical memory.

There is also another class of packets: the *move* packets. This is an unconfirmed remote write that might be used for uncritical data such as video frames or other isochronous traffic.

We will in this model assume that the relative fraction of the different SCI packets sent are equal for all nodes. Thus the fraction of a given packet type is a consequence of the considered application only, not of the node sending the packet. In other words, the following discussion applies to the recent use of SCI as an interconnect for workstations. Originally SCI was designed to be a bus replacement technology where the processor and memory are separate nodes. In this system, the memory node will of course never generate any requests, neither read nor write, but send a lot of responses when serving incoming requests. In a workstation cluster each node has its own local memory and the SCI ringlet may be used to implement message passing through remote memory transactions. It is reasonable to assume the parallel application running on this cluster will be approximately balanced so that all nodes perform roughly the same number and types of transactions.

The parameters for the different packet types are shown in Table 7.1. Observe that the degree of freedom in selecting these parameters is limited by the request-response nature of SCI: If a fraction $f_{R_{\mathrm{req}}}$ request for the read of a packet is sent, then it must be followed by exactly the same ratio of read

Transaction Category	**Packet Type** (T)	**Length** Payload (Bytes)	Total[a] (L_T) (Symbols[b])	**Fraction** (f_T)
Request	Read	0	8	$f_{R_{\text{req}}}$
	Write	16	16	$f_{W_{\text{req}}}(16)$
		64	40	$f_{W_{\text{req}}}(64)$
		256	136	$f_{W_{\text{req}}}(256)$
	Lock Subaction	16	16	$f_{\text{Lsb}_{\text{req}}}$
Response	Read	0	8	$f_{R_{\text{res}}}(0)$
		16	16	$f_{R_{\text{res}}}(16)$
		64	40	$f_{R_{\text{res}}}(64)$
		256	136	$f_{R_{\text{res}}}(256)$
	Write	0	8	$f_{W_{\text{res}}}$
	Lock Subaction	16	16	$f_{\text{Lsb}_{\text{res}}}$
Move & Echo	Move	0	8	$f_M(0)$
		16	16	$f_M(16)$
		64	40	$f_M(64)$
		256	136	$f_M(256)$
	Echo	0	4	—

a The packet header and CRC are included here as appropriate.
b One symbol is two bytes as SCI uses 16-bit wide transmission lines.

Table 7.1. The SCI transactions and symbols

responses to these read requests. The same applies to all requests followed by responses:

$$f_{R_{\text{req}}} = f_{R_{\text{res}}}(0) + f_{R_{\text{res}}}(16) + f_{R_{\text{res}}}(64) + f_{R_{\text{res}}}(256) \tag{7.1}$$

$$f_{W_{\text{res}}} = f_{W_{\text{req}}}(16) + f_{W_{\text{req}}}(64) + f_{W_{\text{req}}}(256) \tag{7.2}$$

$$f_{\text{Lsb}_{\text{req}}} = f_{\text{Lsb}_{\text{res}}} \tag{7.3}$$

Now define f_{req} as the sum of all request fractions, f_{res} as the sum of all response fractions, and f_{move} as the sum of all move fractions:

$$f_{\text{req}} = f_{R_{\text{req}}} + f_{W_{\text{req}}}(16) + f_{W_{\text{req}}}(64) + f_{W_{\text{req}}}(256) + f_{\text{Lsb}_{\text{req}}} \tag{7.4}$$

$$f_{\text{res}} = f_{R_{\text{res}}}(0) + f_{R_{\text{res}}}(16) + f_{R_{\text{res}}}(64) + f_{R_{\text{res}}}(256) + f_{W_{\text{res}}} + f_{\text{Lsb}_{\text{res}}} \tag{7.5}$$

$$f_{\text{move}} = f_M(0) + f_M(16) + f_M(64) + f_M(256) \tag{7.6}$$

From the request-response nature of the protocol, it follows that $f_{\text{req}} = f_{\text{res}}$. The following relations hold between the partial sums of the fractions in the equations 7.4, 7.5, and 7.6:

$$f_{\text{req}} + f_{\text{res}} + f_{\text{move}} = 1 \tag{7.7}$$

$$f_{\text{req}} + f_{\text{res}} = 1 - f_{\text{move}}$$

$$f_{\text{req}} = f_{\text{res}} = \frac{1}{2}(1 - f_{\text{move}}) \tag{7.8}$$

For all types of packets the receiving node issues an echo packet which is positive if the node could receive the arrived packet and store it in the receive buffer, and negative if it could not. A negative acknowledge triggers a retransmission from the sending node. Since all packets are acknowledged with an echo packet, the number of echo packets equals all other packet types. However, as a node never generates the echo packets out of its own initiative, we do not define a special fraction for the echo packets.

As average characterizations of the application defined by the above fractions, we define the mean packet length for each category. This is the weighted sum of the packet lengths where the weights are the fractions generated of a particular length. From the equations 7.4-7.7 we observe that these will not in general sum to unity within each category, hence we must normalize on the appropriate fraction sum for the category:

$$\bar{l}_{\text{req}} = \Big(f_{R_{\text{req}}} L_{R_{\text{req}}} + f_{W_{\text{req}}}(16) L_{W_{\text{req}}}(16) + f_{W_{\text{req}}}(64) L_{W_{\text{req}}}(64) + f_{W_{\text{req}}}(256) L_{W_{\text{req}}}(256) + f_{\text{Lsb}_{\text{req}}} L_{\text{Lsb}_{\text{req}}}\Big) / f_{\text{req}} \tag{7.9}$$

$$\bar{l}_{\text{res}} = \Big(f_{R_{\text{res}}}(0) L_{R_{\text{res}}}(0) + f_{R_{\text{res}}}(16) L_{R_{\text{res}}}(16) + f_{R_{\text{res}}}(64) L_{R_{\text{res}}}(64) + f_{R_{\text{res}}}(256) L_{R_{\text{res}}}(256) + f_{W_{\text{res}}} L_{W_{\text{res}}} + f_{\text{Lsb}_{\text{res}}} L_{\text{Lsb}_{\text{res}}}\Big) / f_{\text{res}} \tag{7.10}$$

$$\bar{l}_{\text{move}} = \Big(f_M(0) L_M(0) + f_M(16) L_M(16) + f_M(64) L_M(64) + f_M(256) L_M(256)\Big) / f_{\text{move}} \tag{7.11}$$

$$\bar{l}_{\text{data}} = f_{\text{req}} \bar{l}_{\text{req}} + f_{\text{res}} \bar{l}_{\text{res}} + f_{\text{move}} \bar{l}_{\text{move}} \tag{7.12}$$

For equation 7.12 we had to cancel the normalizing factors of the previous equations since this is implicitly a sum of all fractions that from equation 7.7 sum to unity.

7.2.2 Packet Generation

Let λ_a be the rate at which node a generates packets to all other nodes measured in packets per cycle. Notice that if $\bar{l}$ is the expected packet length averaged over all packet types, it is an absolute requirement that $\lambda_a \leq 1/\bar{l}$. When equal to this upper bound, the node will on average saturate its output link. This is only possible if there is no bypass traffic on the ringlet as this traffic will have priority on the output link over transmission from the node. Thus, in this extreme case we can expect the transmission queues, one queue for outgoing requests and one for outgoing responses, to build up towards infinity if there is any bypass traffic.

From the definition of λ_a it is evident that the average time between two packets generated is $1/\lambda_a$. Normally, the time intervals between two packets are not all equal to this average, but follow some application specific distribution. As we will apply an average-bandwidth argument here, we will not need to specify this distribution, just require its average to be $1/\lambda_a$ and assume enough buffer capacity at node a to hold all generated packets.

The total arrival rate to the ringlet in number of packets per cycle is

$$\Lambda = \sum_{i=0}^{n-1} \lambda_i \tag{7.13}$$

We observe that the average *throughput* on the ringlet is

$$T = \Lambda \times \bar{l}_{\text{data}} \tag{7.14}$$

7.2.3 Address Distribution

First observe that each node has its individual address distribution, that is the ratio or probability at which it generates packets to the other nodes. However, due to the request-response nature of the SCI protocol there is a dependency between the address distribution of a requesting node and the address distribution of the responding node.

Let $z_{a,b}$ be the ratio of packets generated at node a destined for node b. It is then evident that $\lambda_a z_{a,b}$ is the number of packets generated per cycle at node a heading for node b.

Now let f_{req} equal the sum of all request fractions and f_{res} be the sum of all response fractions. The number of request packets generated at node a for node b is then $\lambda_a z_{a,b} f_{\text{req}}$. Obviously, the number of responses generated at node b for node a must equal the number of packets sent in the opposite direction, thus we have the fundamental identity

$$\begin{aligned} \lambda_a z_{a,b} f_{\text{req}} &= \lambda_b z_{b,a} f_{\text{res}} \\ z_{b,a} &= \frac{\lambda_a f_{\text{req}}}{\lambda_b f_{\text{res}}} z_{a,b} \end{aligned}$$

$$= \frac{\lambda_a}{\lambda_b} z_{a,b} \tag{7.15}$$

Where we used the fact that the total number of response packets generated by an application must equal the number of requests, that is $f_{\text{req}} = f_{\text{res}}$.

As a consequence of equation 7.15, one can only specify half of the address probabilities, the others are given as a function of the specified ones. We also require that

$$\sum_{i=0}^{n-1} z_{a,i} = 1 \tag{7.16}$$

7.2.4 Locality

Central to the performance of the SCI ringlet is the locality of the traffic. Assuming that all requests go to the closest downstream node, we might expect a higher throughput per node than for the case where all traffic is directed to the upstream node bypassing all other nodes on the ringlet.

However, this intuitive picture is complicated by the response traffic: although all initiated requests from a node is for its next neighbor, the responses to these requests have to travel all around the ringlet. There may also be differences in packet lengths. Assume that all requests are `read256` requests, then the responses are much longer than their corresponding requests. In this case it would have been preferable that the traffic was reversed on the ringlet and that the requests went the long way around the ring whereas the responses went only one hop. By obtaining a good communication trace for an application, an intelligent job mapper might try to optimize the locality of the traffic on the ringlet and consequently increase the ringlet scalability.

Now define $\kappa(a,b)$ as the total number of symbol hops caused by a request-response transaction initiated at node a with node b as responder; with the addition for potential move packets. Observe that $z_{a,b} f_{T_{\text{req}}}$ is the probability that a given generated packet is directed to node b and is a request of type T. Introduce L_T as the length of a packet of type T. Consequently we have $\kappa(a,b)$ as the sum of all symbols due to request packets and the sum of all response packets weighted with their travel distances $d_{a,b}$:

$$\begin{aligned}\kappa(a,b) &= d_{a,b}\left(\sum_{T_{\text{req}}} L_{T_{\text{req}}} f_{T_{\text{req}}} z_{a,b}\right) + d_{b,a}\left(\sum_{T_{\text{res}}} L_{T_{\text{res}}} f_{T_{\text{res}}} z_{b,a}\right) \\ &\quad + d_{a,b}\left(\sum_{T_{\text{move}}} L_{T_{\text{move}}} f_{T_{\text{move}}} z_{a,b}\right) \qquad (7.17) \\ &= d_{a,b} z_{a,b}\left(\sum_{T_{\text{req}}} L_{T_{\text{req}}} f_{T_{\text{req}}} + \sum_{T_{\text{move}}} L_{T_{\text{move}}} f_{T_{\text{move}}}\right)\end{aligned}$$

$$
\begin{aligned}
& + d_{b,a} z_{b,a} \left(\sum_{T_{\text{res}}} L_{T_{\text{res}}} f_{T_{\text{res}}} \right) \\
= \;& d_{a,b} z_{a,b} \left(f_{\text{req}} \bar{l}_{\text{req}} + f_{\text{move}} \bar{l}_{\text{move}} \right) + d_{b,a} z_{b,a} f_{\text{res}} \bar{l}_{\text{res}}
\end{aligned}
$$

where $d_{a,b}$ is the distance in number of hops from node a to node b, $\bar{l}_{\text{req}}$ is the average length of a request as given by equation 7.9, $\bar{l}_{\text{res}}$ is the average length of a response from equation 7.10, and $\bar{l}_{\text{move}}$ is given by equation 7.11. Observing that with n nodes on the ringlet, $d_{b,a} = n - d_{a,b}$, and using equation 7.15 we get

$$
\begin{aligned}
\kappa(a,b) &= d_{a,b} z_{a,b} \left(f_{\text{req}} \bar{l}_{\text{req}} + f_{\text{move}} \bar{l}_{\text{move}} \right) \\
&\quad + (n - d_{a,b}) \left(\frac{\lambda_a}{\lambda_b} z_{a,b} \right) f_{\text{move}} \bar{l}_{\text{res}} \\
&= \left[d_{a,b} \left(f_{\text{req}} \bar{l}_{\text{req}} + f_{\text{move}} \bar{l}_{\text{move}} \right) + (n - d_{a,b}) f_{\text{res}} \bar{l}_{\text{res}} \left(\frac{\lambda_a}{\lambda_b} \right) \right] z_{a,b} \\
&= \alpha_{a,b} z_{a,b}
\end{aligned}
$$

$\kappa(a,b)$ represents the amount of traffic on the ringlet caused by messages between the two nodes a and b. Consider the traffic generated from node a to all other nodes on the ringlet as the sum of the $\kappa(a,b)$ over destination nodes b

$$
\begin{aligned}
K(a) &= \sum_b \kappa(a,b) = \sum_b \alpha_{a,b} z_{a,b} \\
&= \sum_{d=1}^{n-1} \left[d \left(f_{\text{req}} \bar{l}_{\text{req}} + f_{\text{move}} \bar{l}_{\text{move}} \right) \right. \\
&\quad \left. + \frac{(n-d) f_{\text{res}} \bar{l}_{\text{res}} \lambda_a}{\lambda_{(a+d) \bmod n}} \right] z_{a,(a+d) \bmod n}
\end{aligned}
\tag{7.18}
$$

Then we may define the average ringlet locality as

$$
\bar{\xi} = \frac{1}{n} \sum_{a=0}^{n-1} K(a) \tag{7.19}
$$

7.2.5 Bypass Rate

Consider a ringlet with n nodes and let the direction of the links be towards nodes with increasing address. Let the nodes be numbered from 0 to $n-1$. We will in this section derive an expression for the expected number of data packets passing through node with index a given the fractions of node a's packets routed to node b. As we are not interested in the type of these packets,

we do not separate between requests and responses, but consider these only as data packets.

Take an example with n say equal to 5; thus the nodes are numbered $0, 1, \ldots, 4$. Consider node $a = 2$. Then for the next downstream node 3, no data packets sent from this node will ever need to pass node 2. This is because all the other nodes on the ringlet can be reached from node 3 without passing node 2. For node 4 on the other hand, node 0,1, and 2 can all be reached without passing node 2, but packets sent to node 3 must pass our tagged node 2. Thus, the contribution to the bypass rate of node 2 from node 4 will be $\lambda_4 z_{4,3}$. Node 0 must send all packets to both node 3 and 4 through node 2, hence it will contribute $\lambda_0 z_{0,3} + \lambda_0 z_{0,4} = \lambda_0(z_{0,3} + z_{0,4})$ to the bypass rate. Finally, observe that node 1 must address all the other nodes except node 2 by sending past node 2. Consequently, its contribution will be $\lambda_1 z_{1,3} + \lambda_0 z_{1,4} + \lambda_0 z_{1,0} = \lambda_1(z_{1,3} + z_{1,4} + z_{1,0})$.

When generalizing this example observe that the addressing probabilities involved can be written as $z_{s,k \bmod n}$ where s is the index of the sending node and k is a summation index running from $a+1$ to some upper limit of the summation which is dependent on the index s relative to the observed node a.

Further notice that the index s starts from the second node downstream from node a, and continues all the way around the ringlet. From this we may write $s = j \bmod n$ for some j running from $a+2$ up to and including $a+n-1$, which is the whole ringlet excluding the node under consideration and its immediate downstream neighbor.

Finally observe that the number of addressing probabilities involved in the summation increases with the distance from the tagged node, and the number of terms in the summation is exactly $(j-1)-(a+1)$, suggesting that $j-1$ can be used as the inclusive upper limit for the summation variable k.

This does indeed hold, and as a justification we return to our example above to see that j attains the values 4, 5, and 6 as we start with $j = a+2 = 4$ since $a = 2$. For $j = 4$, k attains only the start value $a+1 = 3 = j-1$; for $j = 5$, k attains the values $a+1 = 3$ and $4 = j-1$; and for $j = 6 = a+n-1$ we have k equal to $a+1 = 3$, 4 and $5 = j-1$. Having $k \geq n$ is allowed since we use the modulo operator before indexing the addressing probability.

The full rate of data packets passing node a per cycle is consequently given by

$$r_{\text{data},a} = \sum_{j=a+2}^{a+n-1} \left(\lambda_{j \bmod n} \sum_{k=a+1}^{j-1} z_{j \bmod n, k \bmod n} \right) \tag{7.20}$$

7.2.6 Echo Packet Rate

It is not only bypassing data packets that will prevent the node from transmitting. There will also be a stream of bypassing echo packets acknowledging

data packets. Appended to this stream of bypassing echo packets is the stream of echo packets generated at the node itself in response to data packets destined for the node under consideration.

For a node a the rate of received packets will be the sum of packets generated at the source nodes $b = a + 1$ to $b = (a + n - 1) \bmod n$. Recall that the number of packets sent from b to a per cycle is $\lambda_b z_{b,a}$, consequently the arrival rate at node a is

$$r_{\text{receive},a} = \sum_{j=a+1}^{a+n-1} \lambda_{j \bmod n} z_{j \bmod n,a} \tag{7.21}$$

$$= \lambda_{(a+n-1) \bmod n} z_{(a+n-1) \bmod n} + \sum_{j=a+1}^{a+n-2} \lambda_{j \bmod n} z_{j \bmod n,a} \tag{7.22}$$

Here the motivation for the form 7.22 will be apparent when adding this rate with the rate of echo packets passing the node to be derived.

For the echo packets not generated at node a but just passing, we return first to our previous example ringlet of $n = 5$ nodes and once again we will use $a = 2$ as the viewpoint node. Observe that no echo packets from node 3 must pass node 2. From node 4, all echo packets to node 3 must pass node 2. These echo packets are generated as a result of data packets addressed from node 3 to node 4, thus the contribution to the echo rate from node 4 will be $\lambda_3 z_{3,4}$. Node 0 will have to acknowledge all incoming traffic from both node 3 and 4 through node 2, thus its contribution to the echo rate is $\lambda_3 z_{3,0} + \lambda_4 z_{4,0}$. Finally, for node 1 all acknowledge packets must pass node 2 giving a rate of echos of $\lambda_3 z_{3,1} + \lambda_4 z_{4,1} + \lambda_0 z_{0,1}$. When adding these contributions together we get $\lambda_3(z_{3,4} + z_{3,0} + z_{3,1}) + \lambda_4(z_{4,0} + z_{4,1}) + \lambda_0 z_{0,1}$

Observe from this expression that the indices of the λ involved in this final expression can be expressed as $b = j \bmod n$, where j attains the values from $a + 1$ up to $a + n - 2$, both limits inclusive. Further observe that the destination identifier, the second index, of the z in the summations starts off with the value $b + 1$ and runs modulo n to $a - 1$. If we represent this second index with $k \bmod n$ we readily have that k runs from $j + 1$ up to $a + n - 1$, both inclusive. The rate of echo packets passing through node a is therefore given by

$$r_{\text{echo},a} = r_{\text{receive},a} + \sum_{j=a+1}^{a+n-2} \left(\lambda_{j \bmod n} \sum_{k=j+1}^{a+n-1} z_{j \bmod n, k \bmod n} \right)$$

$$= \lambda_{(a+n-1) \bmod n} z_{(a+n-1) \bmod n} + \sum_{j=a+1}^{a+n-2} \lambda_{j \bmod n} z_{j \bmod n,a} + \sum_{j=a+1}^{a+n-2} \left(\lambda_{j \bmod n} \sum_{k=j+1}^{a+n-1} z_{j \bmod n, k \bmod n} \right)$$

$$= \lambda_{(a+n-1) \bmod n} z_{(a+n-1) \bmod n} + \sum_{j=a+1}^{a+n-2} \lambda_{j \bmod n} \left[z_{j \bmod n, a} + \sum_{k=j+1}^{a+n-1} z_{j \bmod n, k \bmod n} \right] \quad (7.23)$$

7.2.7 Output Link Utilization Factor

There are two streams of packets blocking the node from transmitting own data: the stream of bypassing data packets and the stream of echo packets. The rate of the former stream was found in Section 7.2.5 above, and the rate of the latter in Section 7.2.6 above. These rates, equations 7.20 and 7.23, are measured in packets per cycle.

Now we define the *utilization* of the output link from node a due to these two transmission preventing streams as

$$U_{\text{passing},a} = r_{\text{data},a} \times \bar{l}_{\text{data}} + r_{\text{echo},a} \times L_{\text{echo}} \quad (7.24)$$

where $\bar{l}_{\text{data}}$ is the average length of the data packets defined by equation 7.12, and L_{echo} is the length of the echo packets. Notice that this utilization factor is measured in symbols per cycle.

Intuitively we have $U_{\text{passing},a} \leq 1$, where the extreme value $U_{\text{passing},a} = 1$ corresponds to the situation where the bypass traffic totally monopolizes the output link of node a and prevents any transmission from that node. This will not be the case, however, as the SCI standard specifies a flow control allowing all nodes on the ringlet some access to the ringlet (see Chapter 5). We will only require here that the total output link utilization factor obeys the following constraint.

$$U_a = U_{\text{passing},a} + \lambda_a \bar{l}_{\text{data}} \leq 1 \quad (7.25)$$

Hence, we assume that the ringlet is in steady state and all transmission rates are achievable. The transient behavior requires a queueing theory model to be developed for the node, which is beyond the scope of this text.

7.3 Scalability Evaluation

In order to use the bandwidth model developed in the previous section to evaluate the number of nodes that can be attached to a ringlet, one has to specify the application specific parameters used in the model: the fractions of the packet types for the overall system, the packet generation rates and the address distribution for the individual nodes.

7.3.1 Common Assumptions

We will here search overall guidelines for the design of SCI systems, and not link our analysis to any particular application. Most applications running on a workstation cluster exploiting SCI as the interconnect are based on some generic message passing interface, typically MPI. Thus the packet types used will be given by how e.g. MPI uses the SCI protocol to carry messages across. To our knowledge, the most efficient MPI implementation for SCI is *ScaMPI*™ from [9]. This implementation uses only remote write operations, `write64`. The packet length of 64 bytes is used since this is the largest packet size supported by the present hardware. Thus we will assume for the first two experiments that $f_{W_{\text{req}}}(64) = f_{W_{\text{res}}} = 1/2$ and all other packet type fractions are zero.

Assuming an MPI based implementation is also beneficial as it allows us to compare our results with the analysis of Chapter 8. We will also assume that all nodes have identical bandwidth requirements, that is $\lambda_a = \lambda_b = \lambda$. The present SCI adapters are hosted on PCI cards, thus the application has to communicate across the PCI bus with the danger that this bus restricts the achievable bandwidth, or packet generation rate. However, we will here search for the number of supported nodes as a function of their communication requirements.

The raw bandwidth of today's 32-bit wide PCI bus operating at 33 MHz is 132 MByte/s. If we assume that this bandwidth is achievable and the bus is used in both directions to both send and receive packets from the SCI ringlet, we have 66 MByte/s available for transmitting packets. We will evaluate the scalability for several selected PCI bus bandwidths from 25 MByte/s up to 132 MByte/s. In the following, let B_{PCI} denote the available transmission bandwidth, measured in MByte/s.

Now this must be converted into packets per cycle as this is the unit used for the transmission rate λ. By first dividing B_{PCI} by the number of bytes per symbol, we convert the unit to megasymbols per second. By further dividing this by the average data packet length, $\bar{l}_{\text{data}}$, measured in symbols per packet we have the unit packets per second. Finally we need to know how many SCI cycles there are per second. Today's Link Controller LC-2 from Dolphin operates at a clock frequency of 125 MHz, but uses both edges of the clock for transmission. Hence the transmission rate is 250 MHz, corresponding to 250 megacycles per second. Dividing by this factor we arrive at the correct packets per cycles as the required unit for λ:

$$\begin{aligned}
\lambda &= \frac{B_{\text{PCI}}\left(\frac{\text{MByte}}{s}\right)/2\left(\frac{\text{Bytes}}{\text{Symbol}}\right)}{\bar{l}_{\text{data}}\left(\frac{\text{Symbols}}{\text{Packet}}\right)} \\
&= \frac{B_{\text{PCI}}\left(\frac{\text{MSymbols}}{s}\right)}{2\bar{l}_{\text{data}}\left(\frac{\text{Symbols}}{\text{Packet}}\right)}
\end{aligned}$$

$$= \frac{B_{\text{PCI}}}{2\bar{l}_{\text{data}}} \left(\frac{\text{MPackets}}{s} \right) \Big/ 250 \left(\frac{\text{MCycles}}{s} \right)$$

$$= \frac{\text{B}_{\text{PCI}}}{500\bar{l}_{\text{data}}} \left(\frac{\text{Packets}}{\text{Cycles}} \right) \tag{7.26}$$

With different choices for B_{PCI} we will numerically find the number of nodes n that maximizes U_a given by equation 7.25 subject to the constraint that $U_a \leq 1$.

7.3.2 Uniform Ringlet Traffic

Like in Chapter 8 we will assume that the traffic is completely uniform on the ringlet for all pairs of source nodes a and destination nodes b. As a consequence of equation 7.15 we get the following address distribution

$$z_{a,b} = \begin{cases} \frac{1}{n-1} & \text{when} \quad a \neq b \\ 0 & \text{when} \quad a = b \end{cases} \tag{7.27}$$

The corresponding maximum number of nodes the ringlet can support is given in Table 7.2 and shown in Figure 7.1.

PCI Bandwidth (B_{PCI}) in MByte/s	25	35	45	55	**66**	75
Maximum number of nodes	34	24	19	16	**12**	11

PCI Bandwidth (B_{PCI}) in MByte/s	85	95	105	115	125	**132**
Maximum number of nodes	10	9	8	7	7	**6**

Table 7.2. Nodes possible with uniform traffic

7.3.3 Non-uniform Ringlet Traffic

We will in this section analyze the traffic on the ringlet when the traffic is non-uniform. Ideally one could think of a traffic pattern where only two nodes communicate, and then place these two nodes side by side. This is a classical example of extreme locality if only `move` packets are used since the only bypass traffic on the ringlet will be from the echo packets that are almost neglegible. However, in this evaluation of message passing using MPI we will have request-response transactions, leading to a high locality for half of the packets whereas the other half of the packets will have to pass all other nodes on the ringlet.

The corresponding address distribution is

$$z_{a,b} = \begin{cases} \frac{1}{2} & \text{if} \quad b = (a+1) \bmod n \\ \frac{1}{2} & \text{if} \quad b = (a-1+n) \bmod n \\ 0 & \quad\quad \text{otherwise} \end{cases} \tag{7.28}$$

where the second case is a result of the reflexive property of equation 7.15 requiring $z_{a,b} = z_{b,a}$.

Again we get exactly the same results as for the uniform traffic pattern shown in Table 7.2. This comes from the fact that the address distribution has to be symmetric with half of a node's traffic passing all other nodes on the ringlet. If we evaluate the average locality of equation 7.19, we get for a ringlet of 10 nodes $\bar{\xi} = 120$ in both cases. This fact explains why this change in the address distribution did not influence the ringlet scalability.

7.3.4 Changing Packet Lengths

In the last experiment we will investigate the effects of a potential future MPI implementation trying to use longer packages for longer messages in order to reduce the number of SCI transactions per message and, hence, to reduce the message latency. We return to the uniform traffic address distribution and assume that the requests are equally shared between `write64` and `write256` packets.

The immediate result of this change is that the transmission time for an MPI message is reduced at the cost of increased average packet length on the ringlet. As a consequence one may expect fewer nodes to be possible on the ringlet, which is confirmed by the results of Table 7.3. These results are shown graphically in Figure 7.1.

PCI Bandwidth (B_{PCI}) in MByte/s	25	35	45	55	**66**	75
Maximum number of nodes	18	13	10	8	**6**	6

PCI Bandwidth (B_{PCI}) in MByte/s	85	95	105	115	125	**132**
Maximum number of nodes	5	4	4	4	3	**3**

Table 7.3. Nodes possible with `write64` and `write256` packets used

7.4 Discussion

The results presented here on the scalability of a single ringlet with an MPI based application are in line with the results reported in Chapter 8. Our results indicate that the packet size used in the communication contributes inversely to the number of nodes one may have on a single ringlet. We have also seen that the bandwidth available to a given node is roughly of order $O(1/n)$ as intuitively expected.

We have encouraging results from applying this model to the MPI regime, but care must be taken when evaluating other applications perhaps exploiting more of the SCI functionality. The presented model considers only the

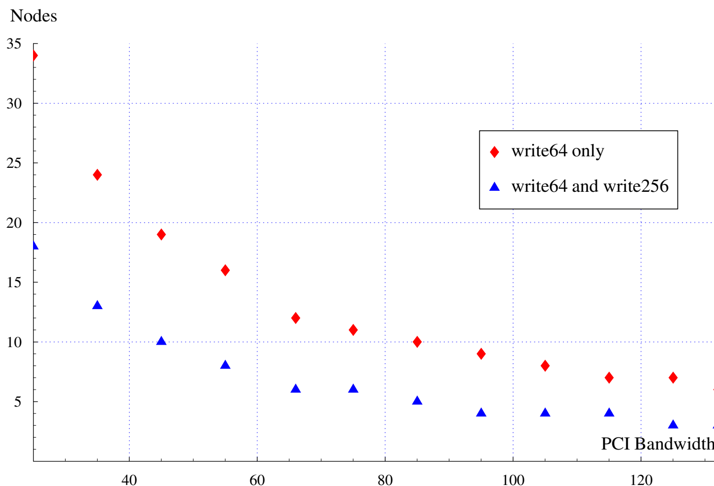


Fig. 7.1. Maximum number of nodes possible on a ringlet under uniform traffic with using firstly only `write64` requests and then an equal mix of `write64` and `write256` requests.

utilization of the ringlet bandwidth, satisfactory for most purposes, but there are other important aspects to address when evaluating the performance of a ringlet:

- The fractions of the different packet types are assumed to be equal for all nodes. Thus if one node always generates packets for another node that only acknowledges or sends short responses back, this is not directly accommodated by the model's parameters. As an example of an application of this kind consider data acquisition in high energy physics where huge amount of data is produced at the sensors and then acknowledged by the various trigger levels. To model this effect in the above equations one will have to reduce the packet generation factor λ for the responding node so that the average number of symbols per cycle is correct.
- The presented model does not consider buffers and buffer overflows as we tacitly have assumed infinite buffer capacity available. In real SCI implementations there are chip-level constraints the system designer are forced to accept. The effect of a buffer overflow in the sending direction is to reduce the amount of data the node is capable of sending, thus the real λ might be less than the one specified. A buffer overflow on the receiving side will cause the sending node to retransmit the packet, thus using the bandwidth twice for the same amount of data. Simulations and measurements reported in [7] have indicated that this retry effect is probably the single major reason for performance degradation in SCI systems.

– Only average steady state bandwidth utilization is modeled here. For systems that are highly oscillating or where the transient conditions do not settle within a reasonable time, the predictions might not be exact. As an example of such a situation, consider a system where the normal traffic is very low, but due to some external event several nodes start an intense communication that settles after some seconds. This kind of system behavior can hardly be accommodated within the existing model parameters.
– The model does not consider the low-level flow control on the ringlet. In equation 7.24 it is possible for the bypassing traffic to saturate a node's output link, leaving no capacity for the node itself to transmit. This situation will trigger a low-level flow control in the real system reducing the allowed bandwidth of the other nodes to a steady state level where they all get about the same access to the ringlet. The net result can easily be accounted for in this model by reducing the selected λ to this post flow-control level; as in the previous comment this is the real steady state conditions for which the model is valid.

Of the model's limitations mentioned above, we suggest for future work that buffer models be included first. This will require a queueing theoretical extension of this work to allow a more complete analysis of the ringlet and the node interfaces.

7.5 Conclusion

We have presented a mathematical model for the bandwidth utilization on an SCI ringlet satisfactorily capturing the major performance effects. This model has been used to evaluate the scalability of MPI based systems achieving figures comparable to results by other researchers. For uniform traffic, the model shows that ringlets up to a maximum size of about 10 nodes are possible given that the output capacity of the nodes is limited by the capacity of the PCI bus.

Future extensions to this model should include queueing models taking the buffer utilization into account in the nodes' interfaces to better evaluate transient conditions and side effects of buffer overflows.

Acknowledgments

This work is supported by Esprit Project No. 25257 (SCI Europe).

References

1. H. Cha, R. Daniel Jr., and A. Knowles. Simulated behaviour of large scale SCI rings and tori. In *Proceedings of the Fifth IEEE Symposium on Parallel and*

Distributed Processing, Dallas, TX, USA, pages 266–274. IEEE Comput. Soc. Press, December 1993.
2. D. B. Gustavson and Qiang Li. The Scalable Coherent Interface (SCI). *IEEE Communications Magazine*, 34(5):52–63, August 1996.
3. IEEE. The Scalable Coherent Interface (SCI), 1992. Standard 1596.
4. S. Kaxiras. Kiloprocessor extensions to SCI. In *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium, Honolulu, HI, USA*, pages 166–172. IEEE Comput. Soc. Press, April 1996.
5. E. H. Kristiansen, J. W. Bothner, T. I. Hulaas, E. Rongved, and T. B. Skaali. Simulations with SCI as a data carrier in data acquisition systems. *IEEE Transactions on Nuclear Science*, 41(1):125–130, February 1994.
6. Ing-Zong Lu and Tien-Fu Chen. Extending SCI on hierarchical directory trees for large-scale multiprocessors. *IEICE Transactions on Information and Systems*, E80-D(4):434–440, April 1997.
7. Knut Omang. *SCI Clustering through the I/O bus: A Performance and Functionality Analysis*. PhD thesis, University of Oslo, 1998.
8. Knut Omang and B. Parady. Scalability of SCI workstation clusters, a preliminary study. In *Proceedings of the 11th International Parallel Processing Symposium, Genva, Switzerland*, pages 750–755. IEEE Comput. Soc. Press, April 1997.
9. SCALI, Hvamstubben 17, 2013 Skjetten, Norway. *ScaMPI User's Guide*, 1998. Version 1.3.0. Available from `http://www.scali.com`.
10. S. L. Scott, J. R. Goodman, and M. K. Vernon. Performance of the SCI ring. *Computer Architecture News*, 20(2):403–414, May 1992.
11. Jens Simon and O. Heinz. SCI multiprocessor PC cluster in a Windows NT environment. *Supercomputer*, 13(2):44–57, 1997.
12. Bin Wu. *The Applications of the Scalable Coherent Interface in Large Data Acquisition systems in High Energy Physics*. PhD thesis, University of Oslo, 1996.

8. Affordable Scalability Using Multi-Cubes

Håkon Bugge, Knut Omang

Scali AS
Hvamstubben 17
N-2013 Skjetten, Norway
Email: {hob,knuto}@scali.no

8.1 Introduction

This chapter presents an analysis of the scalability of Scali systems. A Scali high-performance server consists of compute nodes, interconnected by a high-speed, low-latency SCI interconnect. The topology addressed in this paper will be direct networks based on r-ary f-cubes, or multi-dimensional tori, also called multi-cubes. The focus is on the scalability of a specific implementation of the SCI architecture, namely the PCI to SCI adapter boards from Dolphin Interconnect Solutions [5]. The adapter boards are enhanced with more than one SCI link controller (LC) in order to increase the number of supported dimensions (or fan-outs).

We show how the SCI ringlets and the internal bus in each adapter limit scalability of the interconnect, and how the two relate to each other. The internal bus (the B-Link) is used to take packets between different dimensions and between each dimension and the PCI interface towards the local node.

The resulting analysis can serve as a guide to select the right topology for a given system size. We discuss how the interconnect scales with respect to the amount of traffic each node can generate, which is limited to the bandwidth of a single PCI bus, and argue that these topologies scale to 512 nodes using state-of-the-art technology.

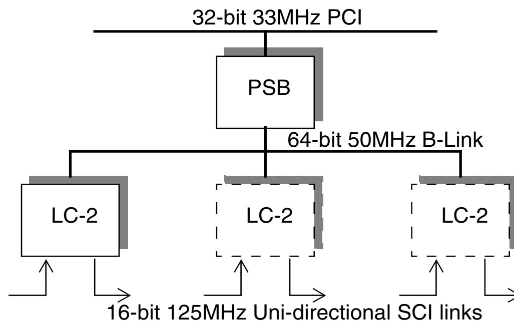


Fig. 8.1. Block diagram of PCI to SCI adapter board

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 167-175, 1999
© Springer-Verlag Berlin Heidelberg 1999

8.2 Interconnect Overview

This chapter covers systems which use a 32-bit, 33 MHz PCI bus [8] as the attachment point to the interconnect. The bridge functionality from PCI to SCI is handled by the PSB (PCI to SCI Bridge chip) [3]. This bridge translates PCI memory operations to SCI transactions and vice versa. Please note that the PSB chip does not contain any SCI link, but interfaces to B-Link (Backside Link for SCI link chips) [2]. B-Link is a multi-master split request-response bus where the packet format is a superset of the SCI packet format. The translation from B-Link to SCI physical layer is performed by the LC-2 (Link Controller 2) [4]. The block diagram of the PCI to SCI adapter board is shown in Figure 8.1. The gross bandwidths of the buses and the SCI links are listed in Table 8.1. Here we see the PCI bus as a restriction; however, both B-Link and the SCI links might be burdened by traffic to/from other nodes.

Bus	Width (bits)	Frequency (MHz)	Gross Bandwidth (MByte/s)
PCI	32	33	133
B-Link	64	50	400
SCI	16	125	500

Table 8.1. Gross bandwidth of buses and links

8.3 Methodology

To evaluate scalability we will see how the different parts of the interconnect restrict the bandwidth in the case where all nodes communicate with all other nodes. The first step in the analysis will be to determine the efficiency of the different parts of the interconnect. Then we will look at the impact the SCI links and the B-Links have on bandwidth as a function of topology and system size. The restriction in interconnect bandwidth which stems from the PCI bus is independent of the size and topology of the system. This is because the PCI bus only restricts the rate with which the interconnect can deliver packets to a node and the rate at which the node itself emits packets to the interconnect.

The Scali system uses SCI *nwrite64* request packets from the requester node (initiator in PCI terminology) to the responder node (target). The responder node writes the 64-byte payload to memory and signals the completion of the transaction by sending a *resp00* back to the requesting node. The 80-byte large *nwrite64* packet will use 11 cycles on B-Link. The 16-byte *resp00* packet will consume 3 cycles on B-Link. A single cycle is required on B-Link to switch between different masters (bus turnaround).

When a *send* packet (either request or response) is taken off an SCI ringlet, an 8-byte echo is returned to the node which injected the *send* packet into the ringlet. Furthermore, each packet will be followed by two 2-byte idles. The SCI specification dictates the use of one idle after each packet, but LC-2 uses two. The efficiency of B-Link and SCI link *nwrite64/resp00* traffic is calculated in Table 8.2.

B-Link (cycles)		SCI links (bytes)	
Req	11	Req	80
Bus turnaround	1	Req.idle	4
Rsp	3	Req.echo	8
Bus turnaround	1	Req.echo.idle	4
Total (cycles)	16	Rsp	16
Total (bytes)	128	Rsp.idle	4
Payload (bytes)	64	Rsp.echo	8
Efficiency (%)	50	Rsp.echo.idle	4
		Gross Size	128
		Payload	64
		Efficiency (%)	50

Table 8.2. Efficiency of B-Link and SCI links

B-Link restricts the available bandwidth of a given node; the available bandwidth is shared between all traffic going to the node, the traffic being generated by the node, and the traffic which uses the switch constituted by the attached LCs. The maximum number of packets which have to pass a single B-Link is called *hot-B-Link*.

The SCI links which constitute the interconnect fabric, do not provide each sink-source pair with a set of dedicated SCI ringlets. Hence, the total number of packets which has to flow through a single ringlet segment will restrict the available bandwidth. The total number of packets passing through a single ringlet segment is called *hot-link*.

The term *hot-link* is defined in [6]. In their analysis, the adapter internal interconnect is assumed to be a true switch. The term *hot-queue* is used to describe the number of packets traversing through a particular queue in the adapter-internal switch. Since the LC-2 based adapters used in Scali systems are based on a bus-based switch, i.e., the B-Link, we had to define the *hot-B-Link* term to precisely define the adapter internal traffic rate. The analysis in [6] is based on the assumption that all nodes transmit N packets, i.e., each node sends a packet to itself. In our case we assume nodes only send to the $N-1$ other nodes on the interconnect.

8.4 Analysis

When all nodes communicate with all other nodes simultaneously, a single node will issue $N-1$ packets. The bandwidth available to a node will be the $N-1$ packets issued, divided by the hot traffic, multiplied by the effective bandwidth.

Assume a regular multi-cube where each node is connected to f (bi-directional) SCI ringlets (i.e. the adapter board is populated with f LC-2 chips), and that each ringlet is connected to r nodes. The number of nodes is then $N = r^f$. An example of a 3D torus, where $r = 4$ and $f = 3$ is given in Figure 8.2. Note that each line in Figure 8.2 indicates an SCI ringlet, as illustrated by one ringlet in each of the three dimensions. A multi-cube consists of

$$R = f * r^{f-1} \tag{8.1}$$

ringlets. Since each ringlet has r segments, the total number of link segments, S, in a multi-cube is

$$S = r * R = r * f * r^{f-1} = f * r^f = f * N \tag{8.2}$$

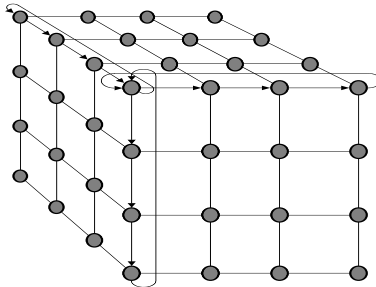


Fig. 8.2. 3D torus, $r = 4$ and $f = 3$

8.4.1 "Hot-Link" Analysis

The average number of ringlet segments traversed by a packet within a single ringlet can be calculated. Assume that a node in a ringlet sends one packet to each of the $r-1$ other nodes on the ringlet. The $r-1$ packets will traverse $1, 2, 3, ..., r-1$ segments correspondingly. The average number of ringlet segments visited is then

$$\frac{1 + 2 + \ldots + r - 1}{r - 1} = \frac{(r-1) * r}{2 * (r-1)} = \frac{r}{2} \tag{8.3}$$

Considering a minimal routing in the multi-cube [1], we can calculate the probability for a packet to traverse a ringlet in each of the f dimensions. The number of packets sent from a node which has to traverse a ringlet in a particular dimension, is the total number of nodes minus the node sharing the same coordinate in the given dimension. The number for nodes sharing one coordinate in a given dimension is $r^{f-1} = r^f/r = N/r$. Thus, each packet has a probability of

$$\frac{N - N/r}{N - 1} = \frac{N * (r - 1)}{r * (N - 1)} \tag{8.4}$$

for traversing a ringlet in each of the f dimensions. Since the total number of packets sent is $N * (N - 1)$, we can express the *hot-link* traffic as

$$\begin{aligned} HotLink &= \frac{N * (N-1) * f * N * (r-1)/(r * (N-1)) * r/2}{f * N} \\ &= \frac{N * (r-1)}{2} \end{aligned} \tag{8.5}$$

8.4.2 "Hot-B-Link" Analysis

The number of packets flowing through a B-Link is the number of packets sent by the node plus the number of packets received by the node plus the number of packets which switch dimensions at the node, i.e. the packets which are taken off one ringlet and inserted into a ringlet in another dimension. The number of packets sent and received by the node is $2 * (N - 1)$. Of the total number of packets sent from a node, $N - 1$, some portion will only traverse one dimension, another portion will traverse two dimensions, and so on, and the last portion will traverse all f dimensions.

Let P_d indicate the number of packets which traverse d dimensions. Assume $\mathcal{N}$ being an f-digit number using radix r initially containing all zeros. P_d equals the number of permutations of $\mathcal{N}$ having d digits different from zero. There exists $f*(f-1)*(f-1)*\ldots*(f-(d-1)) = f!/(f-d)!$ positions for d digits in $\mathcal{N}$. Since the d digits might be permuted within $\mathcal{N}$, we divide by $d!$. The number of d-digit numbers having all digits different from zero is $(r - 1)^d$, and P_d can be expressed as

$$P_d = \frac{f! * (r - 1)^d}{d! * (f - d)!} \tag{8.6}$$

The *hot-B-Link* traffic is the sum of the packets sent by a node plus the number received by the node plus the weighted sum of P_d. A packet which traverses d dimensions will switch dimension $d - 1$ times, hence we use this as the weight. The *hot-B-Link* traffic can then be expressed as

$$HotBLink = 2 * (N - 1) + \sum_{d=2}^{f} (d - 1) * P_d \tag{8.7}$$

8.5 Results

As discussed in the previous section, the maximum bandwidth available to a node in a multi-cube is restricted by the ringlets and the B-Links. The maximum available bandwidth is then the smallest of the two, i.e., the maximum bandwidth available to a node is the minimum of the restrictions imposed by the *hot-link* and the *hot-B-Link*. The *hot-link* factor is illustrated for two topologies in Figure 8.3.

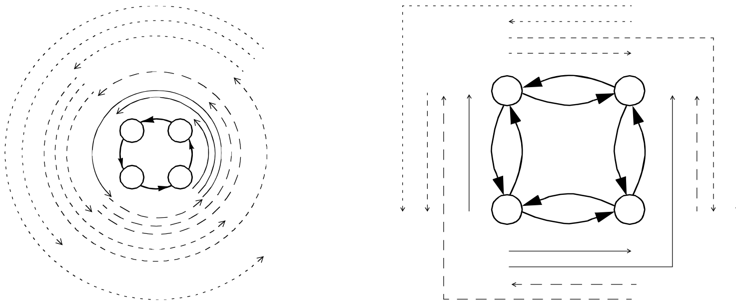


Fig. 8.3. *Hot-link* illustration; 4-node ringlet (left) with a *hot-link* factor of 6, and a 2-ary 2-cube (right) with a *hot-link* factor of 2

The bandwidth available to a node as restricted by the *hot-link* is illustrated in Figure 8.4. The ringlet's inability to scale is clearly visible and we see that more dimensions in the topology cease the restriction imposed by the SCI ringlets.

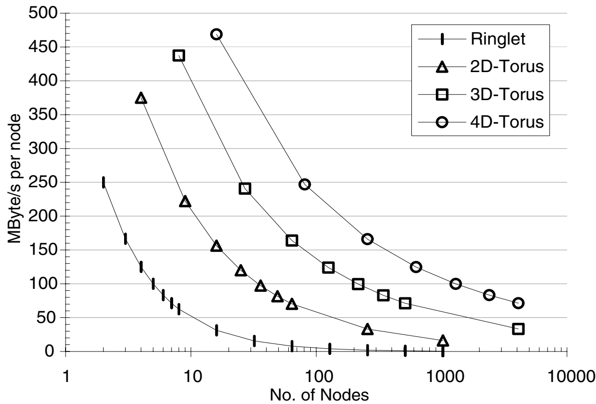


Fig. 8.4. Available bandwidth per node as restricted by the hot-link

The restriction in bandwidth imposed by the *hot-B-Link* is illustrated in Figure 8.5. The bandwidth per node for the one-dimensional ringlet is constant with respect to system size, since the B-Link traffic is proportional to the system size. However, for more than one dimension, the B-Link will limit the available bandwidth both with increased number of dimensions as well as increased system size.

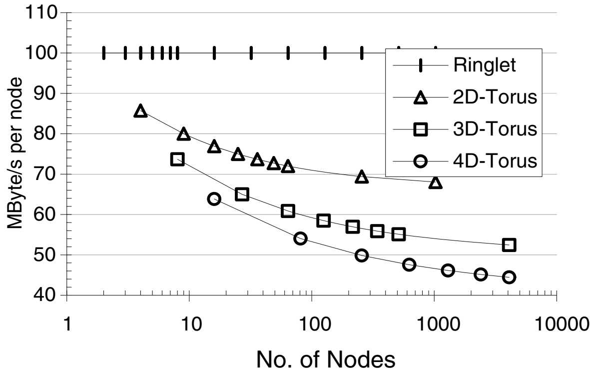


Fig. 8.5. Available bandwidth per node as restricted by the hot-B-Link

The bandwidth available to a node is restricted by the minimum of the bandwidth constrained by the *hot-link* and the *hot-B-Link*. This is plotted in Figure 8.6. The knee on the curves is the transition point where B-Link limitation equals the limitation imposed by the SCI links. Left of the knee, the hot-B-Link limits the per-node bandwidth, whereas the *hot-link* is the limit for a larger number of nodes.

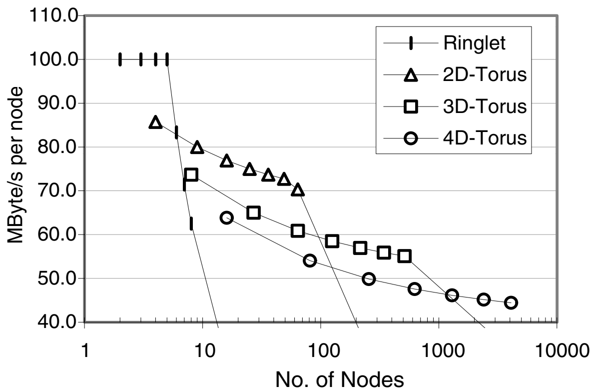


Fig. 8.6. Available bandwidth per node as restricted by the either the *hot-link* or the *hot-B-Link*

From Figure 8.6, the optimum fanout can be chosen for a given system size. For systems with 6 or fewer nodes, a single ringlet is the best choice. A two-dimensional torus is the best choice for up to 64 nodes. For 64 nodes, the ringlet size restricts the bandwidth to 70 MByte/s in a 2D torus, whereas the B-Link restricts the bandwidth to 61 MByte/s in a 3D torus of the same size. Minimum latency is increased when dimension switching is necessary. The time taken to switch directions once on the PCI/SCI adapter board for a 64-byte transaction is around 700 ns while bypass latency per node within a ring is around 25 ns [7]. Clearly, a 2D torus is the best choice. As the number of nodes passes 100, the 3D torus becomes the best choice, and this holds true for up to 1000 nodes. A 3D torus having 512 nodes will sustain a bandwidth of 55 MByte/s per node.

It is interesting to see how the interconnect bandwidth relates to the I/O bus bandwidth. The PCI bus used in these Scali systems has an efficiency of about 75% on the write burst cycles to/from the SCI/PCI adapter board. Since the PCI will be burdened with traffic in both directions, PCI will restrict the available bandwidth per node to half of the sustained peak in one direction, i.e., roughly 50 MByte/s. Since the bandwidth provided by the SCI interconnect is higher, the scalability in terms of bandwidth is linear up to 512 nodes (assuming a 3D torus) for Scali systems.

8.6 Conclusions

The interconnect bandwidth for various multi-cube topologies has been analyzed. For the implementation of the SCI architecture used in Scali systems, we have shown linear scalability in terms of interconnect bandwidth. We will use these results as a guide to select the right fanout for building Scali systems of a particular size. The analysis should also be valuable for others who want to build or plan the purchase of SCI based tori or simply want to understand the dynamics of such systems.

Acknowledgment

The authors are grateful for the help in deriving equation 8.6 given by Per Berge Johannesen.

References

1. W. J. Dally and Ch. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, 36(5):547–553, May 1987.

2. Dolphin Interconnect Solutions. *A Backside Link (B-Link) for Scalable Coherent Interface (SCI) Nodes*, draft 2.4 edition, September 1995.
3. Dolphin Interconnect Solutions. *PCI-SCI Bridge Functional Specification*, version 3.01 edition, November 1996.
4. Dolphin Interconnect Solutions. *Link Controller LC-2 Specification*, version 1.03 edition, October 1997. Preliminary version.
5. Dolphin Interconnect Solutions Inc. *PCI-SCI Cluster Adapter Specification*, version 1.1 edition, May 1996.
6. R. E. Johnson and J. R. Goodman. *Interconnect Topologies with Point-to-Point Rings.* Technical Report 1058, CS Department, University of Wisconsin – Madison, December 1991.
7. K. Omang. Performance of a Cluster of PCI Based UltraSparc Workstations Interconnected with SCI. In *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing, Las Vegas, Nevada*, volume 1362 of *Lecture Notes in Computer Science*, pages 232–246. Springer-Verlag, February 1998.
8. PCI Local Bus Specification, Revision 2.1.

Part IV

Device Driver Software and Low-Level APIs

To facilitate secure, transparent, user-level access to the distributed shared memory (DSM) of the SCI hardware, a variety of setup and resource management tasks have to be performed. As for other I/O devices, such tasks are carried out by the operating system, via the SCI device driver code. Whereas communication over SCI is lightweight and efficient, the management tasks enabling this are considerably more complex than those required, e.g., for a mainstream network, as pointed out in Chapter 9.

The functionality that has to be covered by an SCI device driver includes: exporting, distributing, importing, and mapping of DSM segments used for communication; providing protection for, and locking of, pages comprising the shared segments; handling of persistent transfer errors, link and node failures, and erroneous behavior of processes; raising and delivering interrupts; exception handling; and programming and controlling DMA engines.

Chapter 9 reports on these challenging issues and their solutions in two SCI device drivers, those from Dolphin Interconnect Solutions and Scali A.S. Moreover, the approach chosen and the lessons learned in porting these two drivers to Linux are described, providing interesting insights into the complexity of low-level SCI software and into the experiences made with the Dolphin SCI hardware.

A different perspective is given in Chapter 10 which describes the SCI Physical Layer API (SCI PHY-API), currently being in the final phase of standardization as part of the family of SCI standards (IEEE P1596.9). The standard does not aim at defining full SCI device driver functionality, but rather attempts to specify a lean software layer abstracting the hardware DSM and enabling its use with minimum added overhead, such that real-time applications are not constrained from a performance point of view. The hardware abstraction is not specifically tied to SCI, but may be applied to other DSM interconnects as well. It is interesting to compare this standard functionality with the device drivers reported in Chapter 9.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 177-178, 1999
© Springer-Verlag Berlin Heidelberg 1999

A prototype implementation of SCI PHY-API has been done in the context of a data acquisition system in a high energy physics application, described in Chapter 23. That chapter also introduces another low-level SCI API called the SISCI API which is, however, more application-oriented in that it abstracts both the hardware and the device driver software.

9. Interfacing SCI Device Drivers to Linux

Roger Butenuth, Hans-Ulrich Heiss

Operating Systems and Distributed Systems Research Group,
University of Paderborn, Germany
email: {butenuth, heiss}@uni-paderborn.de
http://www.uni-paderborn.de/cs/heiss/

9.1 Introduction

The ultimate goal of the SCI standard is to support memory coupling of different SCI nodes, where remote memory accesses are handled entirely by the hardware. So one may ask: Why do we need a driver? At least using the memory of a 'usual' computer needs no driver, so where is the difference? The answer is simple: There is no difference in *using* the memory, SCI memory looks just slower for the processor whenever the SCI hardware has to fetch a remote cache line.

The task of an operating system is to provide a virtual view to all processes: they all see their own virtual version of the machine, more or less perfectly isolated from other processes. The current situation is to do some of this virtualization in hardware (e.g. address translation and protection by the MMU), and some in software (e.g. time-slicing, file handling). Which parts are done in hardware mostly depends on efficiency considerations. Emulating a MMU in software is possible, but rather slow.

The MMU does only one part of the memory handling, translation of virtual to physical addresses and signaling faults to the operating system. The other part of management is still done by the operating system. The memory management is usually tightly coupled with file system caches, a way to avoid expensive copy operations of large memory blocks.

What is changed by putting SCI hardware into a node? On the hardware level, there are two changes: a node can access local physical memory and parts of the 64-bit SCI memory space visible in its physical address space. Access to all memory areas is still handled by hardware, but there is the need to manage how remote memory is accessed. Today's operating systems are not prepared to handle this, making drivers necessary which handle proper allocation of local and remote memory resources. This puts many management tasks into the driver, making it complicated compared to other drivers that access only a 'simple' device[1].

[1] The Dolphin SCI driver source is more than three times larger than the largest SCSI driver in the Linux kernel.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 179-190, 1999
© Springer-Verlag Berlin Heidelberg 1999

9.2 Layers of Functionality

9.2.1 Address Spaces

Before starting with the possible layers of SCI device drivers, a short look at some of the address spaces involved seems to be necessary. The memory in SCI systems can be seen from different points of view. First of all, the 64-bit SCI address space is split into 48-bit node address spaces and a 16-bit node index (Figure 9.1). It is most likely very sparse populated. In a typical system neither all 2^{16} nodes nor all 2^{48} addresses within the nodes will be used. The next view is the physical address space of the nodes. This can be 32 bits (e.g. Intel x86) or 64 bits (e.g. DEC Alpha) wide. Both imply restrictions on the mapping to the SCI address space. A 64-bit local address space is able to map the whole SCI address space, but in the other direction SCI with its 48-bit node address space cannot map the whole local space of all nodes. This is no real restriction, as long as only 48 bits of the local space are used. On the other hand, a 32-bit local address space puts many more restrictions on the system: it can map only a small portion of the SCI address space and most likely not all physical memory available in the system. Today, SCI systems with more than 4 GByte total physical memory are quite common. In these systems, a process can only map a fraction of the whole physical memory. This is just the opposite situation as on local systems, where the virtual memory is larger than the physical memory!

9.2.2 Levels of Hardware Abstraction

SCI drivers in kernel space can provide different abstraction levels to the user level software on top of the kernel. We will discuss the different levels before taking a closer look at two implementations, the Linux version of the Scali driver [8] and the Dolphin driver [3]. Both were ported at the University of Paderborn in the project ‘Arminius’ [2].

A solution using no interrupts can live entirely in user space. It is sufficient to map all hardware addresses into the virtual address space of a process and to manipulate the hardware from there. The approach has some drawbacks which make it infeasible in most cases, but it gives an easy way to start development without the trouble of writing code that is part of the kernel. This approach requires that all processes accessing the SCI hardware of the node have root privileges and gets around all security barriers in the kernel. It may be an option for a dedicated system that controls some other hardware connected by SCI. In a multi-user environment like a computing center, this is not an option. Another problem is the allocation of physical memory from user space. This is not supported in standard Linux kernels, but one could reserve some of the physical memory at boot time for this purpose to solve the problem.

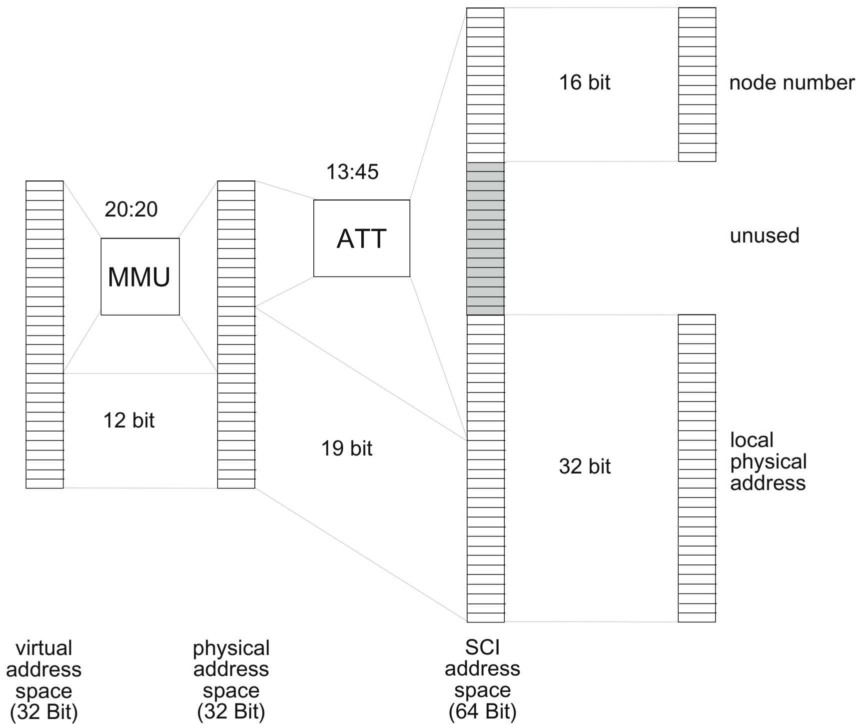


Fig. 9.1. Different address spaces and mappings between them.

Instead of mapping the whole SCI adapter card registers into a user process' address space, one can write a small kernel level driver with access functions to the features of the SCI interface. This enables the driver to do some access control, avoiding that only root processes can use the interface. With this type of interface it is possible to forward interrupts to user level, an approach to avoid polling. A user level library could be used to implement a standard compliant interface, e.g. SISCI. Even this approach lacks resource management in the kernel, complicating cleanup after termination of processes using the driver. The whole information about allocated resources is stored in the processes, either in user code or in a library. A cleanly terminating process can free all its resources, a crashing process cannot. Together with the memory footprint of the process, all allocation information vanishes. It should be obvious that this is no option in a multi-user environment with 'untrusted' programs.

Safe multi-user operation makes proper resource management in the driver necessary. The driver has to maintain information which processes—local and remote—use the resources. Especially keeping track of remote resources adds a lot of complexity and requires communication between drivers on different nodes. Whenever a process owning a resource terminates, all processes on

other nodes have to be notified. In some cases the notification is difficult to achieve, but it must be guaranteed that the other processes cannot access the resource after it has been relinquished. Some early drivers did not consider this problem, making it possible to overwrite memory on a remote node that was no longer allocated by the driver. This is possible without MMU intervention on the remote system because it is a direct memory access from the SCI hardware. Eventually the kernel on the system will crash.

9.2.3 Resource Management

From now on we concentrate on drivers which are more than just wrappers around register access functions because the distributed resource management is one of the interesting parts of the drivers. SCI memory is managed in multiples of pages; the Scali driver calls a piece of memory a *chunk*. A chunk is located on one node and consists of physical memory that is never paged to disk. Paging of SCI memory would be possible, but every time a page is swapped out to disk, all other nodes sharing this page would have to be informed. Although this would be possible, it is difficult to realize. With respect to current prices for memory and SCI adapters it seems to be much better to put more memory into the nodes rather than to go to the trouble of paging SCI shared memory to disk.

Another issue is raised by the capabilities of the SCI hardware in the nodes. The PCI/SCI adapter card from Dolphin [3] maps parts of the SCI address space to an address window within the PCI address space. The 'mini-MMU' responsible for this uses 512-kByte pages. This means there are only 2048 pages per GByte of the address window. The amount of memory a node can import through SCI is limited by the size of the window and its utilization. The mismatch between the page sizes of today's CPUs and the SCI page size can result in bad utilization of the address window. In case the imported memory is scattered on the exporting node, the worst case may happen, where only one MMU page (e.g. 4 kByte on Intel x86 systems) of a 512 kByte SCI page is used. With a 1 GByte address window it is possible to import 2048 * 4 kByte = 8 MByte in this case.

To avoid the worst case scenario, a chunk should consist of only few continuous parts (subchunks) of memory. A size of 512 kByte for the subchunks would be sufficient for optimal usage when all subchunks are aligned to 512-kByte boundaries in physical memory. Unfortunately, most modern operating systems have no support to handle such large pieces of physical memory. Demand paging needs only one page at a time, resulting in a scattered main memory where it is not easily possible to find large free contiguous parts.

There are different possibilities to solve this problem. First, simply ignore it. The result is bad usage of precious space in the physical address window and limited access to the SCI address space. Second, one can reserve some memory for SCI usage and withdraw it from the memory management of the operating system. We selected this solution in the Linux version of the

driver. This requires a small patch on Linux, the so-called *bigphysarea patch*. It is not very elegant and should be seen as an interim solution, as long as the third possibility has not been exploited. The third solution would require major changes to the memory allocator of the operating system, in order to avoid the scattering of pages throughout the physical address space.

Any piece of memory allocated by the driver in this manner must get a proper identifier to make it addressable by any client of the driver. Care must be taken not to free this memory until all users have released it, otherwise stale references with danger of overwriting memory of other users will occur. It is useful to have a possibility to withdraw a resource from a client. Without such a withdraw mechanism, a client—from a local or remote node—could lock a piece of memory indefinitely.

The Scali driver uses two different identifiers: one generated by the driver when the memory is allocated and one that can be chosen by the allocator. It consists of the *node number* where the chunk resides and two integers, called *user identifier* and *chunk identifier*. Associating these numbers with a chunk makes it 'visible' to other processes. Removing the association from the chunk withdraws it from all clients. This may need communication between the drivers on different nodes.

9.2.4 Virtual Mapping

Allocating a piece of memory is only the first step to make it accessible for a process, additionally it has to be *mapped* in its address space. The steps for a mapping differ in the local and remote case. The local case is quite simple. On an `mmap` call to the driver, programming the local MMU is all that is necessary. The problem is to tell the driver with the parameters of the `mmap` system call which chunk should be mapped. Only the file descriptor and an offset into the file can be used for this purpose. The Dolphin and Scali drivers use these two parameters in different ways.

The Dolphin driver requires to create a new file descriptor for every chunk of memory. This results in a one-to-one relation between file descriptors and chunks, removing any ambiguities. In contrast, the Scali driver can associate more than one chunk with one descriptor. The chunks associated with the same descriptor are identified by the upper bits of the offset parameter of the `mmap` system call. The 32 bits in the offset field are divided into some bits for the chunk identification and some bits for the 'real' offset within the chunk. Thus, the number of chunks per file descriptor and the chunk size are limited. How many bits are reserved for these parts is a compile time parameter of the driver.

To map a remote chunk, another step is necessary: before the MMU can map some physical addresses into the virtual address space of a process, the part of the SCI address space has to be mapped into the local physical address space of the node. This is done by the Address Translation Table (ATT) in the SCI adapter. The corresponding driver operation is called 'import remote

chunk'. Before the driver can set the entries in the translation table, it has to know the physical address of the chunk to be imported. For this purpose it sends a request containing the chunk identifier to the exporting node. On receipt of this message, the node answers with the physical address and registers the new client in its data structures. In this way, it can inform all clients whenever the chunk is withdrawn. After the physical mapping is established, a handle is given back to the caller. For the Scali driver, this handle contains the upper bits of the offset needed for the `mmap` call.

9.2.5 Robustness

Throughout the text some of the possible errors have been mentioned. This section gives a summary and shows what has been done in the drivers to cope with the errors and which errors are simply ignored. Another issue is security, currently ignored by both the Dolphin and the Scali drivers.

The services of a driver can be used from within the kernel by another driver or from user space by application processes and libraries. In the first case, one may rely on the correctness of the parameters, at least in some levels. In the second case, there should be no assumptions about the value of any parameter. The driver (and the kernel) should survive any call with any parameter values. Parameter checking here is more difficult than in most other drivers because it has to consider the local state and the states of other nodes in the system. One call, e.g. a withdraw operation, can affect all other nodes.

Parameter checking is done by the Dolphin and Scali drivers; at least in theory it should not be possible for any application to crash the driver or operating system. Due to the complexity of the drivers however, one can still expect bugs in both of them. A crashing process is handled differently by the drivers. In the Scali driver, all resources owned by the process are destroyed immediately. In case of memory chunks, this has the same effect as a withdraw operation, so all other processes sharing this piece of memory can no longer use it. One exception exists when there is a second process on the same node sharing the chunk locally. In this case a reference counting mechanism delays freeing the chunk until this process releases it. Dolphin handles this in a different way, holding the chunk until all other processes release it.

Another topic is how failing nodes are handled within an SCI system. The impact from one crashing node to the rest of the system depends on whether the node exports or imports pieces of memory. A failure on an importing node is a problem when the exporter does reference counting on the shares from all other nodes. In this case, the reference counter is never decremented. To free the resource, the exporter needs to be notified of the failure in some way. The Dolphin driver does this by alive checks of all nodes it shares resources with.

A crashing exporter is more critical. When a hardware failure occurs, where the node goes down completely, all memory operations involving this node will fail on the hardware level, which can be recognized by other nodes. A software failure where all hardware is still functional is more critical. In this case, accesses to shared memory are still possible. The problem arises after reboot. The freshly booted node has no knowledge about the exported memory, so this may be used in a total different way. In the long term, this will result in another failure. Dolphin handles this problem by the alive checks in the driver, Scali ignores it on the driver level and delegates it to user level software.

9.3 Why Linux?

There are drivers for Solaris and NT; so one may ask why we ported both drivers, first the Dolphin driver, then the Scali driver, to Linux. As an operating system research group, we believe it is important to have the possibility to look into the details of the operating system and to change parts thereof. For NT the source code is very difficult to obtain. The same applied for Solaris when we started with SCI; this has changed meanwhile, since the Solaris source is available to academic sites. Nevertheless, the code of Linux is documented [1, 7] and there are many people around who can help whenever questions arise. The only drawback of Linux was the limited support of SMP systems; its single kernel lock is not very efficient when processes are often in kernel mode. This has changed with the 2.2 kernel series with its fine grain locking in the kernel. On the other hand, the Linux kernel is quite efficient, e.g. the TCP/IP latency over Fast Ethernet is only half of that of Solaris on the same hardware.

The modular design of the kernel made only minor changes in the kernel code necessary: the *bigphysarea patch* and exporting some more symbols at the module interface. The SCI driver itself is a kernel module, which made development easy. It was not necessary to compile a new kernel or to reboot the system for every new compiled driver.

After working with the Dolphin driver on Linux for more than one year, a Scali system was installed in Paderborn; see Chapter 21. This system contains a modified Dolphin SCI adapter (second Link Controller), so we could not use the Dolphin driver on that system. This was the reason to start all over again and to port the second driver. This driver was designed to be portable. Building on the experience from porting the Dolphin driver it was possible to have the first test version after about three weeks. More than two months of tuning, debugging, and testing followed, resulting in a robust driver, even more robust than the original Solaris version. (We know of programs that cause operating system crashes with the Solaris version but run fine with the Linux version.)

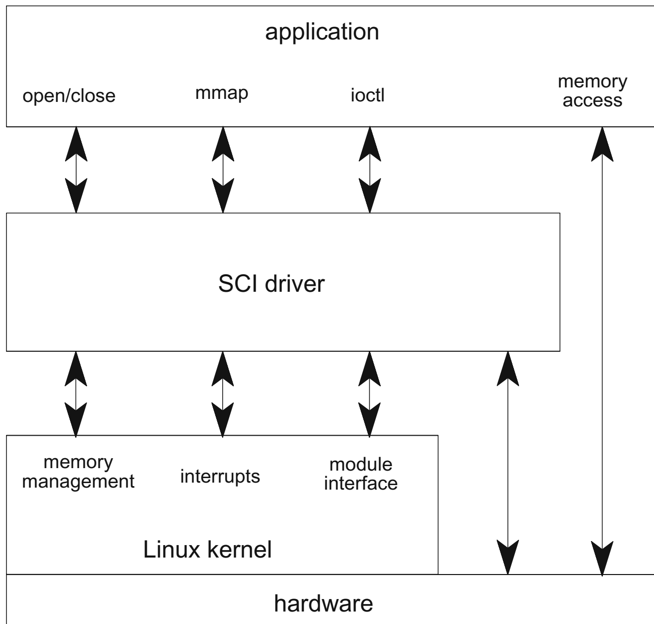


Fig. 9.2. Interfaces of the driver.

9.4 Interfaces of the Driver

The driver interacts with four other entities (see Figure 9.2): the hardware, the operating system, user processes, and its counterparts on other nodes. The interface with the operating system is more complicated than in many other drivers due to its deep involvement with the memory management.

9.4.1 Hardware

The hardware interface of the driver is more or less the same for the Solaris, NT, and Linux version of the driver. There are abstractions for all the registers on the card to handle the fact that SCI is big endian and the host can be little (e.g. x86) or big (e.g. SPARC) endian. On initialization some interaction with the operating system is necessary. The SCI card must be visible in the kernel address space and a callback function for interrupts from the card has to be established. Interrupts are generated for various events, some of them are errors, arrival of messages from other SCI nodes, and user generated interrupts. The source of the interrupt must be recognized by the callback function.

9.4.2 Linux

Linux has a small and elegant interface for device drivers, it is very similar to drivers implemented as modules and drivers integrated into the kernel. On loading, an initialization function of the driver is called. This and a cleanup function are the only entry points into the driver known to the kernel at link time. The driver announces all other entry points by calling back the kernel with function pointers as arguments. For all drivers needing an interrupt one of them is the interrupt handler. The same interrupt can be shared by several drivers. On interrupt, all handlers are called and the drivers have to recognize whether their hardware has generated it. Most other callbacks are announced together with one register function, where the driver declares itself responsible for one major device with a set of 'methods', like `open`, `close`, `read`, `write`, `mmap`, `ioctl`, etc. Thus, a driver can be seen as an object with methods. It has not to provide all methods; default methods of the kernel are used then.

The most complicated part is the management of virtual mappings into the address spaces of user processes. A mapping can come to existence in different ways: First, by the process itself when it invokes the `mmap` system call. Second, by a `fork` system call where one process with its address space and all other parts is duplicated. Removal of a mapping is handled by a reference counter. It is not removed on a `close` of the file descriptor. Unix semantics allows a mapping to exist beyond the associated file descriptor's lifetime. Even when the file descriptor no longer exists, the mapping is still accessible. It is destroyed by a call of `munmap` or on termination of the process.

The removal of a mapping upon a withdraw request from another node is difficult to implement. The request is received by an interrupt handler, so everything has to be done in the handler before the answer is sent back to the requester. This situation—an interrupt handler removes a mapping—has not been foreseen by the designers of the Linux memory management. Instead, it is required that the process whose memory mapping has to be removed, is currently running and therefore can take care of this task. In most cases one can expect that another process is running when the interrupt occurs. This situation requires to use a different technique: the mapping is not removed, instead the code in the handler marks all pages as non-accessible. This ensures that no memory of the withdrawn mapping is visible to the user process. Any attempt to read or write in that area is detected by the MMU and results in a page fault. The page fault is forwarded by Linux to the device driver (calling the page fault handler function), which returns with the information 'page not available'. Eventually the kernel raises a `SIGSEGV` to the process. A better solution would be to inform the process immediately after the withdraw operation, not on access of one of the addresses in the segment, but there is no standardized mechanism in Unix for this type of event.

9.4.3 User Processes

The interface from user processes to kernel level device drivers is limited to a small set of functions; the more important ones are: `open`, `close`, `read`, `write`, `lseek`, and `mmap`. These are sufficient for most standard drivers, but they offer no possibility to control all the parameters needed to communicate with the SCI driver. There is one 'escape function' that can accomplish this: `ioctl` (abbreviation for I/O control). It takes one integer with a command code and a pointer to memory as parameter. This pointer can reference a structure with additional parameters, setting no restrictions on the number and size of parameters. The disadvantage is the inconvenient interface. One has to pack all parameters into a structure, call the function, and unpack everything from the structure. This is avoidable by wrapper functions above and below the '`ioctl` layer'. A user level library is on top of the interface and a set of functions in the driver breaks up the `ioctl` call to calls of the individual functions.

Such a small library is still a very low-level interface. For parallel shared memory programming more functionality is needed:

- Controlled start and termination of parallel programs.
- Creation of distributed segments with control where the memory is physically located.
- Synchronization support, at least barrier synchronization and semaphores.
- Global operations, like global sum, min, max, and Boolean operations on various data types.
- Coordinated interaction between processes, e.g. something like signal/wait.

A lot of this functionality is available in two libraries, Yasmin (Yet Another Shared Memory INterface [9]) and SMI (Shared Memory Interface, [4]). They work on different driver interfaces, Yasmin on top of the Scali driver, SMI on top of the Dolphin driver. Additionally, both can use Unix shared memory on SMP machines. This is useful for development and tests or whenever there is no SCI hardware available.

Yasmin and SMI now have their own start mechanisms (early versions of SMI used an underlying MPI for that purpose), but both of them are quite simple: a list of machines is given in a file which is read by the start mechanism that creates the processes on the nodes. In Yasmin this is done with the secure shell (`ssh`). In a multi-user environment, there should be a more comfortable solution, e.g. CCS (see [5] and Chapter 26). Yasmin will use CCS as its start mechanism as soon as it is available for Linux.

9.4.4 SCI Drivers on Other Nodes

An important interface of a driver is that to its counterparts on other nodes, which makes it a distributed driver. This is not common practice in the Unix

environment, where most distributed services are implemented as user level daemons[2].

The Dolphin card contains a mailbox system which can be used to deliver small control messages between nodes (see Chapter 3). The messages are stored in a ring buffer and their delivery is signaled to the driver by an interrupt. The size of one message is fixed to 64 bytes, which is sufficient for this task. The Dolphin and Scali driver use this messaging system in quite a different way.

The approach in the Scali driver is connectionless, conceptionally like an RPC service. A memory chunk is more or less handled like a server that replies to requests from clients. There is not much state information in the 'chunk server'; of course there must be some, e.g. for sending the withdraw messages mentioned earlier.

The Dolphin driver, with its safety measures, uses a much more complicated, connection-oriented protocol. The driver periodically checks each node to which it has a connection, whether it is still up and running, and increments its heartbeat counter. Thus, it always knows when some resources are no longer available on a logical level even when they are available on the physical level[3]. All these fault tolerance provisions make the driver quite complicated and large. It remains an open question whether or not this would be better done on a higher level in user space.

9.5 Conclusions

When we started our work with SCI, we underestimated the complexity of the driver. The first statement from Dolphin about the size of their driver, about 50,000 lines of code, immediately resulted in the question whether they had said 5,000. But, as it turned out, they had not! — The Scali driver is much smaller. It could be designed simpler because Scali does not address the fault tolerance problem in the driver (their target are parallel machines, not fault tolerant servers) and because they have no workarounds for the bugs in the first SCI cards. The complexity of the drivers is the reason for their long development time. Early versions contained many bugs. Together with bugs in the early SCI cards they made SCI unstable. This has changed now, Rev. D cards with LC-2 and the current drivers are quite stable.

The work with two different drivers and the possibility to look into the operating system dependent parts of the code has shown similar problems in all combinations, one of them allocation of large, contiguous blocks of

[2] There are exceptions, e.g. the NFS server is sometimes implemented as a kernel thread for performance reasons.

[3] Even when the operating system on a remote node has crashed, the memory may still be reachable through the SCI adapter. This can be fatal when the node comes up again and the memory is still in use by the remote node.

physical memory. In Linux we could solve that problem by a small kernel modification.

The large common code base for the different systems has saved a lot of implementation work. Unfortunately, Dolphin and Scali could not agree on a common driver code base, doubling a lot of work. Many research groups working with the Dolphin cards could get access to the source code of the Dolphin driver. On the other hand, the Scali driver is usually available in binary format only. We think the small, closed group of software developers is an obstacle to the wide distribution of SCI. The open policy of Myricom has resulted in a lot of freely available communication libraries for Myrinet cards [6].

References

1. M. Beck et. al. *Linux Kernel Internals.* Addison-Wesley, 1997.
2. R. Butenuth, H.-U. Heiss. *Project Arminius Homepage.* http://www.uni-paderborn.de/fachbereich/AG/heiss/arminius/
3. Dolphin Interconnect Solutions. *The Dolphin SCI Interconnect.* White Paper. Dolphin. http://www.dolphinics.com.
4. M. Dormanns, W. Sprangers, H. Ertl, T. Bemmerl. A Programming Interface for NUMA Shared-Memory Clusters. *Proc. High Performance Computing and Networking (HPC)*, pages 608-612, LNCS 1225, Springer, 1997.
5. A. Keller, A. Reinefeld. CCS Resource Management in Networked HPC Systems. *Proc. Heterogeneous Computing Workshop (HCW'98)* at IPPS, Orlando, 1998.
6. Myricom. http://www.myricom.com
7. A. Rubini. *Linux Device Drivers.* First edition, O'Reilly & Associates, 1998.
8. S. J. Ryan. *The Design and Implementation of a Portable Driver for Shared Memory Cluster Adapters.* Research Report no. 255, Department of Informatics, University of Oslo, December 1997.
9. H. Taskin. *Synchronisationsoperationen für gemeinsamen Speicher in SCI-Clustern.* Diploma thesis, University of Paderborn, December 1998.

10. SCI Physical Layer API

Volker Lindenstruth[1], David B. Gustavson[2]

[1] Institute for High Energy Physics, Schröder Str. 90,
69120 Heidelberg, Germany
email: ti@ihep.uni-heidelberg.de
http://www.ihep.uni-heidelberg.de/

[2] SCIzzL/Santa Clara University
1946 Fallen Leaf Lane
Los Altos, CA 94024-7206
email: dbg@SCIzzL.com
http://www.SCIzzL.com/

10.1 Introduction

The IEEE SCI standard defines a shared memory interconnect from the physical layer to the transport layer. However, no standard software layer is defined. One might argue that in a distributed shared memory environment little software is required because once a distributed shared memory (DSM) system is set up, all accesses can be performed by directly reading/writing from/to the appropriate target addresses. This is the main advantage of the distributed shared memory architecture. It results in the lowest possible message passing latency and transaction overhead. No procedures or system services need to be called in order to exchange data between different nodes in the system.

Although there is no software required to perform the DSM data exchange, there is a fair amount of software infrastructure necessary, for example, to create an appropriate shared memory segment, to export it into the global shared address space or to import that global shard memory segment into the local address space of another process. If DMA is to be used, appropriate structures need to be set up to control the DMA hardware. DSM transactions can fail resulting in exceptions that have to be handled.

In order to be able to decouple DSM hardware and software development, the IEEE P1596.9 working group was formed that focused on the architecture of a generic DSM API for SCI. This API was to be hardware independent and to support any operating system. The goal of this working group was to define a DSM software layer, with minimal added overhead, that generalizes the software interface to necessary hardware functionality such as initialization of address maps, DMA block moving, error and exception handling and the like. The requirement of minimal overhead and latency drove this software interface to become more a hardware abstraction layer than what is typically considered a driver.

Transaction overhead and latency are very important features in real-time applications, where distributed shared memory systems are widely used. One

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 191-205, 1999
© Springer-Verlag Berlin Heidelberg 1999

particularly demanding field of use is the high-energy-physics community where very large amounts of data are processed by very large clusters of processors at very high transaction rates. For example a detector system STAR (Solenoidal Tracker At RHIC) [1]) contains a multiprocessor system capable of handling event or transaction processing rates of up to 100 kHz. Such transaction rates emphasize the importance of low overhead and latency with respect to the multiprocessor interconnect. In the case of STAR, SCI was chosen as interconnect standard. Other high energy physics (HEP) systems and requirements are presented in Chapter 23. In order to allow the development of the necessary DSM software independent of the availability and specifics of the necessary SCI hardware, the P1596.9 SCI Physical Layer API working group defined a software standard which is detailed in the remainder of this chapter.

It should be noted that, despite the SCI context, this API is useful for any DSM scenario. Especially, cache coherent and non-coherent environments are equally supported.

10.1.1 Scope of the Standard

There are two competing goals in defining a low level API such as the SCI Physical Layer API. One is to absorb as many hardware peculiarities as possible into the API in order to simplify the higher level software interface. The other is to keep it as simple as possible in order to allow the highest possible performance and not to constrain any possible hardware architecture such as multiple rings connected with bridges, switched networks, multidimensional meshes or a combination thereof.

In the field of high-speed real-time DSM multiprocessor systems unnecessary software overhead cannot be tolerated. Therefore, the scope of this software standard was drawn such that it does implement all necessary hardware abstraction functions but avoids any additional functionality that would increase the overhead.

Another class of functionality is related to system level issues. For example a DSM multiprocessor cluster is required to be initialized and maintained, potentially implementing fault tolerance features such as failover. Other issues are global address resolution in a given system, since a given node in the system needs to be able to find out about the physical location of a given specific shared object. In the case of heterogeneous systems, endianness conversion/correction protocols could become necessary. There are many more related issues. However, they all have in common that they require knowledge about the given hardware architecture and require additional functions and protocols to be implemented.

Many discussions were held during the various working group meetings about what functionality should be included in this API. It soon became obvious that there are various high level software standards, which satisfy different requirements, that could benefit from a common DSM hardware

abstraction layer, but that it is also impossible to serve all fields of use at once. For example, MPI is a message passing standard that could still benefit from the low message passing overhead and latency that SCI provides. In the case of MPI, some interface or middle layer software is required to match the IEEE API to the MPI API. However, such middle layer or meta driver has to be implemented only once in order to gain complete hardware independence for any SCI based MPI system if the IEEE API is used as underlying software standard.

Given this standard being submitted to IEEE and its scope as a hardware abstraction layer, it did not seem appropriate to try to define, for example, a standard address resolution protocol or an SCI network topology implementing failover functionality within the scope of this standard. It is left to the higher level software to implement those functions. However, the SCI Physical Layer API implements standard methods required to support the implementation of such functionality.

10.2 SCI Physical Layer API Architecture and Features

This chapter gives an overview of the SCI Physical Layer API. For a detailed reference, please refer to the draft standard document itself. Figure 10.1 below shows its overview.

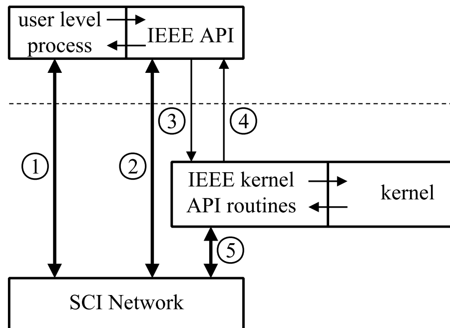


Fig. 10.1. A sketch of the SCI Physical Layer API Architecture. The various data paths (1) through (5) are detailed in the text.

A user level process interfaces to the API in order to perform all procedural functions. Depending on the operating system, the API may consist of user level routines and kernel level routines. For example, in the case of initialization and mapping calls, the user level API routines would call the protected kernel level API routines (3) which would perform any authentication and security checking in order to ensure that no unauthorized physical

access is granted. Responses to API procedure calls and any exceptions are returned to the user level API routines (4). Initialization such as cold start and configuration of the SCI interfaces' address maps is typically also executed in the API kernel routines (5). This is necessary in order to ensure that no user program can directly access the security relevant address translation tables in the interface. The SCI Physical Layer API requires functionality which is not supported by the hardware to be emulated in software. For example, should the interface not support DMA functionality, the API is required to implement that as programmed I/O. This functionality is expected to reside in the user level part of the API (2). Once a shared memory is set up, the user level process can perform shared memory transactions without involvement of the API as indicated by path (1). This feature allows the smallest overhead and latency.

Two general classes of transactions are supported as indicated in Figure 10.2 below.

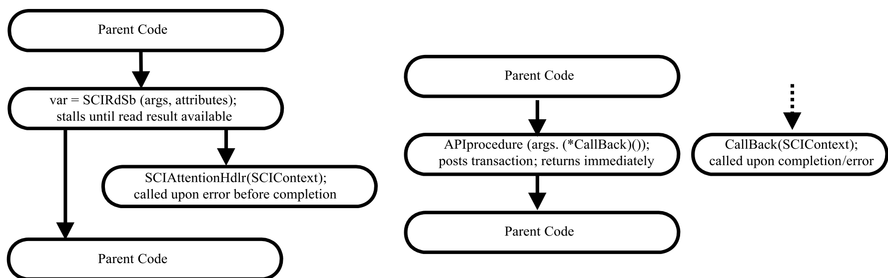


Fig. 10.2. A sketch of a synchronous transaction (left) with a potential exception, and a sketch of an asynchronous transaction (right)

Synchronous transactions such as read transactions (Figure 10.2, left) return after valid read data is available. This results in potential stalling of the host processor. If an error occurs, an exception is fired (refer to Section 10.2.1). The second class of transactions are asynchronous transactions, which return as soon as possible without awaiting the completion of the actual SCI transaction. This is illustrated in the right half of Figure 10.2. Upon completion of the asynchronous transaction, a specified call-back procedure is executed. The body of this procedure can be used to implement whatever synchronization method is desired. Examples of asynchronous transactions are posted writes or DMA transactions. In the case of posted writes the call-back procedure would typically act only if an error occurs. In the case of a DMA transaction, the call-back would notify the host about the DMA completion status.

Before an SCI transaction can be executed, a certain amount of setup and control is required. For example, address translation tables may need to be configured appropriately in order to map an SCI address region into the

process address space. This is implemented based on windows. A window is a contiguous address region with a defined set of default transaction attributes and configurations. Those default transaction attributes define whether write transactions may be posted, a window is write protected, or write transactions to this given window are executed as broadcast, etc. The appropriate address translation setup of both the operating system and the SCI interface is part of the configuration.

10.2.1 Exception Handling

The SCI Physical Layer API interfaces to external hardware and consequently requires a method for asynchronously handling exceptions, such as link failures or asynchronous conditions such as failing posted writes, DMA completion, etc. Since this standard is required to be operating-system independent, a very simple method is supplied for implementing asynchronous attention handlers. It is based on the definition of a context structure, allowing the complete description of the state of the SCI hardware and software interface. For an asynchronous attention condition, a call-back procedure is executed which allows user supplied implementation of the exception handler. The appropriate procedure is supplied as a procedure pointer by the higher software layers and acts much like an interrupt handler.

Some transaction specific exceptions cannot be traced to the calling process(or). For example, a posted write transaction exception may occur long after the write request is completed. Other posted writes may have been executed in the meantime. In this case it is not possible to determine the appropriate transaction specific call-back procedure. Therefore, a global exception handler is implemented which will be executed in such a case. However, the global attention handler can only identify the type of condition, based on the context structure. Therefore, in order to debug and trace these conditions, posted transparent writes must be disabled.

Asynchronous transactions such as the chained-mode DMA use call-back procedures to provide a tool to synchronize with the host program. This is done by using whatever synchronization method is supported best by the given operating system (signals, events, semaphores etc.).

In order to allow implementation of checkpoints, a synchronization transaction is provided, which stalls the calling process until all pending transactions have completed and all related potentially pending transaction handlers were executed, or a specified timeout expired.

10.2.2 Endianness

SCI defines the byte endianness for the physical data transfer to be address-invariant. The endianness of any given data type is not defined within the SCI scope. Further it is not possible to implicitly determine the endianness

of any given data type within the SCI context. Implicit data conversions cannot be performed without requiring additional protocol layers that would inquire about the endianness feature of any node in the system, or without encapsulating any shared memory data set with an appropriate descriptor. Therefore, endianness conversions are considered outside the scope of the SCI Physical Layer API. All data objects are viewed as bags-of-bytes, handled in an address-invariant fashion, as in the SCI context.

10.3 Supported Data Types

The standard set of integer data types defined in ANSI C has become insufficient. Further, the exact size of `short`, `int` or `long` is not specified and may vary from system to system. Given the environment of a distributed shared memory, it is very important to be able to specify the exact size of a data object platform-independently. To eliminate that problem the IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors [4] was created, which defines a set of standard data objects such as `Doublet` (two-byte object) or `Quadlet` (four-byte object).

All data types used in the SCI API are based on [4]. In order to support 64-byte and 256-byte transactions two additional data types were added (`Blocklet` for 64-byte packets and `Qblocklet` for 256-byte packets). In case a compiler does not support the native shared-data formats [4], the specified data types have to be mocked up by `typedef` statements in the standard include file. Standardized preprocessor flags allow conditional code segments for cases where the compiler does not support all data sizes.

Obviously there are derived data types such as structures defining various objects such as the DMA chain entries, the chain mode DMA status list, messages, or the API state in the case of an exception. However, they all are composed of the set of basic data types described above.

10.4 Miscellaneous Procedures

A canonical set of setup, initialization, and control procedures is defined. Multiple, possibly different, SCI interfaces are supported in one host. Therefore, during initialization and setup, the API connects to a simple database where it can inquire about the availability of SCI interfaces. The client then selects one or several SCI interfaces and receives the appropriate list of API entry points for the given hardware. Now the interface can be opened and configured.

Default exception handlers need to be defined in the case of failing transactions that cannot be traced back to the originator (such as posted writes). These exception handlers are interface specific and are set up when the interface is opened.

10.5 Address Translation Model

In order to be able to perform a shared memory transaction on a remote node, the appropriate local address space needs to be directly accessible to the remote processor. No software is involved on the local node when the remote node accesses the given address space region. The advantage of this implementation is that locally no CPU cycle is used in performing the transaction. The only burden on the local node is the consumption of memory bandwidth due to remote accesses. However this feature is very dangerous if the exported memory regions are not chosen carefully. For example, system critical address regions such as the system interrupt vector table could be overwritten by a remote node if they were made accessible to the SCI address space. It would be very difficult to debug such a system. The solution to this problem is the segmentation of the address spaces into local, protected address space and global address segments, which are exported into the SCI address space.

Figure 10.3 below shows an example of a typical scenario where four processes on two processors communicate across the SCI global address space.

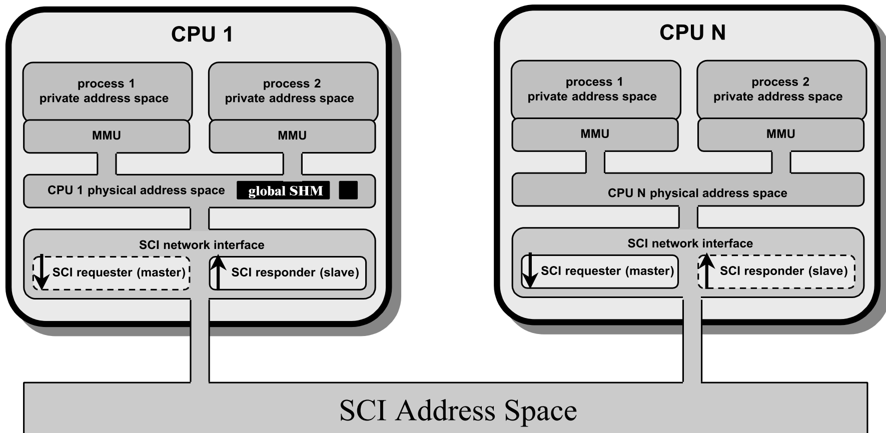


Fig. 10.3. Architecture of two nodes' communication across an SCI network

In the example in Figure 10.3, processor 1 runs several processes, which all may have their own private virtual address space. Note that some operating systems, such as VxWorks, OS9 and other typical real-time kernels, avoid the virtual memory concept but rather implement the virtual and physical address space to be identical, maybe with some added access control. A defined shared memory segment resides somewhere in the physical address space of processor 1. Access to this memory segment is accomplished by mapping the memory segment into the processes' address spaces, which is typically done by setting up the appropriate MMU page tables.

In order for this memory segment to be made available in the SCI global address space, called the transport space, the local SCI interface needs to be configured to map requests to a specified SCI address window to the local address space corresponding to the shared exported local shared memory. This enables other SCI nodes to directly access any data word within the given shared memory. There is no constraint where the given shared object is to reside in the SCI address space.

The shared memory segment may be fragmented. There are various possible scenarios dealing with this issue. One is to simply export the fragmented memory segment as such into the SCI transport space and to implement a higher-level protocol that distributes the fragmentation information to all nodes involved. Another scenario is to hide local memory fragmentation by programming the appropriate address translation tables of the SCI interface such that the locally fragmented shared memory is exported into a contiguous SCI address window. The advantage of this scenario is that every shared memory segment can be assumed contiguous in the transport space. The SCI Physical Layer API standard does not make any assumptions or restrictions here, but rather allows a local contiguous memory region to be exported into the SCI address space. In the case of a fragmented local memory and the first scenario, the appropriate export procedures have to be executed once per segment.

In order for a remote node to access the given shared memory segment, it needs to configure the SCI interface requester to map the appropriate SCI address window(s) into the local physical address space, which then is mapped into the appropriate process' address space.

Figure 10.4 shows a sketch of the various address spaces involved in a distributed shared memory system. The appropriate API calls are also indicated. The fragmented shared memory segment (SHM) resides in the physical memory of processor 1. It is mapped into the private address spaces of the local processes. Remote access is granted by exporting it into the SCI transport space of the given local node. Access to this segment by a remote node requires the given SCI address window to be mapped into the processor's local address space and this region to be mapped into the appropriate process' private address space. This functionality is called importing of an SCI address segment.

The shared address segment may also be an I/O region. This feature is useful for real-time systems, where I/O devices (for example, trigger boxes and readout devices) are controlled directly by a remote node. From the software point of view there is no difference since in both cases an address segment is handled. The only software-visible difference would be that caching or even only read-ahead gathering would be disabled in the case of the I/O segment.

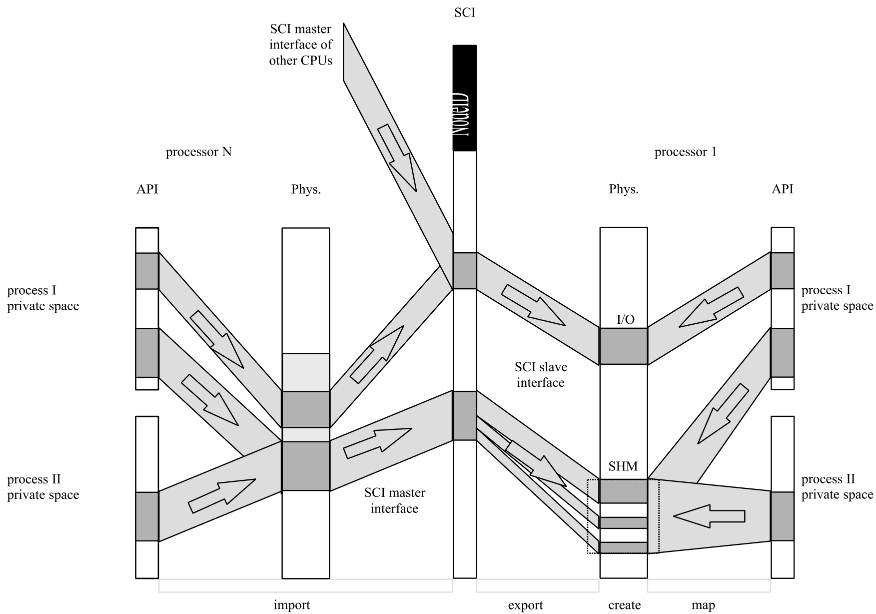


Fig. 10.4. A sketch of the various address spaces and mappings involved in a distributed shared memory system

10.5.1 Global Object Identifier

During the discussion above it became obvious that there can be a large number of shared objects within a distributed memory system. Shared memory objects may be created and referenced by different processes and processors. This requires a method for unambiguously identifying the given shared resource, which should be system-wide unique. One system-wide unique identifier is the 64-bit SCI address. However, this address also reflects the physical location of this object. Therefore, any changes to the architecture of the system would result in changing object identifiers. Another example is the case of fault tolerant systems where critical shared data structures may have to exist as redundant copies in order to be able to simply re-map shared global objects in case one critical node becomes unavailable. Based on a unique global object identifier, which is independent of its physical address, every node can inquire about the potentially changed location of a shared resource and receive the appropriate physical location (SCI address).

In order to keep the identification of the resource as general and as flexible as possible, it was implemented as a 128-bit ID. Since it is only required for setup and control procedures, there is little overhead involved with such a large ID. With 128 available bits for identification of a given shared memory object, its ID can be made unique system-wide.

10.5.2 SCI Global Address Resolution

There is no restriction as to where a given shared object is to be exported in the SCI address space. In order to be able to map a defined global shared memory object based on its ID, some address resolution method needs to be implemented. There are two major scenarios. One defines a static SCI address map which is known to all nodes. A more generic and much more flexible scenario implements an SCI global address resolution. However, defining a standard address resolution protocol would be beyond the defined scope of the standard. On the other hand, all basic methods needed to implement such functionality are provided in form of a standard, basic message passing API (refer to Section 10.9).

Based on that message passing functionality, the global address resolution can be implemented by the following convention. Each node exporting or importing at least one shared memory segment is required to listen (refer to Section 10.9) at the defined address resolution port. A message sent to the specified port contains the ID of the memory segment in question. Upon receipt of such a request, a defined response is generated containing the physical SCI base address of the object.

Here the requester still needs to know the appropriate SCI node ID, which exports the shared memory. This can be performed by simply polling all available SCI nodes. A more elegant implementation is a distributed shared memory address resolution server which maintains knowledge of all available shared memories and can be contacted by any node seeking address resolution. Therefore, if a given node receives an address resolution request for a shared memory segment not defined locally, but known, it shall return the corresponding SCI base address in a proper response message. In the case that it does not know the given object, it shall return another host ID that might act as shared memory address resolution server.

Those primitives allow implementation of the necessary address resolution functionality without defining a specific protocol but merely providing all necessary mechanisms.

10.6 Shared Memory Transactions

Once the shared memory system is set up, data can be read or written by directly accessing the appropriate addresses, which does not require any programming interface to be defined. There are appropriate procedures and data types defined (refer to Section 10.3) allowing transactions of defined size from 1 byte to 256 bytes. This enables one to perform SCI transactions within assignment statements.

However, it is not possible to specify further arguments within an assignment statement. A specific group of shared memory transactions is defined that allows one to specify further arguments such as the transaction size,

transaction attributes, and a specific exception handler to be used only for the given transaction.

A special group of transactions within the class of shared memory transactions are lock transactions. SCI supports a set of atomic transactions that are executed on the target node. In order to provide a programming interface for those atomic transactions, appropriate functions are defined, where the lock command and lock argument are passed in the function's argument list, and the result of the atomic transaction is returned to the caller as in the case of any other shared memory read function.

```
[1]     #define SCIWrByte(Adr, Data)  (*((*Byte)Adr) = Data)

[2]     SCIWrByte           (address, data8bit);
[3]     SCIWrSb             (address, bytecount, datapointer);

[4]     data = SCIRdSb      (address, bytecount);
[5]     lock = SCILockSb    (address, bytecount, datapointer,
                             attentionhandlerpointer, lockcommand);
```

Fig. 10.5. Some examples of selected byte commands

Figure 10.5 shows some shared memory code examples. Lines 1 and 2 show a particular feature of the IEEE API. The single-byte write transaction (`SCIWrByte`) could also have been implemented as pointer assignment, rather than using the stated procedure call. This procedural semantic does not impose any unnecessary overhead since it is translated into the appropriate zero overhead assignment statement as indicated by the preprocessor macro in line 1, which is typically part of the standard include file. There is an appropriate data type and procedure defined for all standard data sizes ranging from one byte to 256 bytes. Although remote shared memory transactions could be implemented as a pointer assignment statement in the source code, the stated form is recommended. This functionality allows one to port even a shared memory based application onto an architecture that does not support shared memory transactions. In that case procedures like `SCIWrSb` would be truly procedures implementing the appropriate put and get functionality. Such implementations would result in potentially significantly larger overhead. However, shared memory software written according to this programming paradigm can be run without modification on a non-shared memory platform, provided the appropriately modified API is made available.

Line 3 in Figure 10.5 shows a write transaction that uses a non standard object size such as 15 bytes. The appropriate read transaction (`SCIRdSb`) shows the appropriate counter part. It returns a defined 16-byte object, which is filled with `bytecount` bytes returned from the target node. The atomic transaction shown in line 5 (`SCILockSb`) returns the result of the lock request and receives as argument the target address, object size, a pointer to the lock

argument of appropriate size and the lock command. A specific attention handler or call-back procedure is specified as indicated.

10.7 Packet Transactions

Most known SCI interfaces do not support the full set of defined SCI transactions. Therefore an optional packet transaction is implemented that allows one to send an SCI packet to a defined target node. The packet argument is a defined data structure. A possible response is identified based on the transaction ID, which is assigned by the API, and forwarded back to the host program using the specified call-back procedure.

10.8 Block Transactions

The SCI standard supports moving large blocks of data by implementing block-transfer hints within the packet headers, indicating to intermediate bridges the intent to send a stream of packets. These intermediate agents may prefetch data based on the DMA controller's announced intent. In order to utilize this functionality, the SCI client software must be able to communicate the intent to move blocks of data. On the other hand, if the available SCI interface hardware supports DMA functionality, it permits the block move with very little software overhead. There are many different chain-mode DMA engines defined. This standard tries to abstract the DMA programming interface without constraining any existing hardware or incurring large additional software overhead.

Figure 10.6 shows a sketch of the chain-mode DMA descriptor architecture. It uses the IEEE P1285 [5] concept, but removes the specifics with respect to the mass storage framework. All chain descriptors are adjacent in memory, forming a DMA chained command list block. Each chained command is a defined fixed length 32-byte structure. However, there is a defined command allowing implementation of several disjoint chained command list blocks. The number of these blocks, or chain command descriptors is only limited by the amount of available memory. The end of a chained DMA list is defined by a specific command as shown in Figure 10.6.

There is also an optional array of status descriptors defined. The number of status descriptors is fixed during the transaction. The DMA engine will use the defined status descriptor field like a cyclic buffer. Therefore, the last N status entries are available for review by the caller. Each chain command may cause a status entry depending on the setting of a group of mode bits within the structure. In the event of an error, a status entry is always written. The specified exception handler may be triggered as well.

Some SCI hardware may not support the specific chain command format. Therefore, a specific procedure is provided that translates the input chain

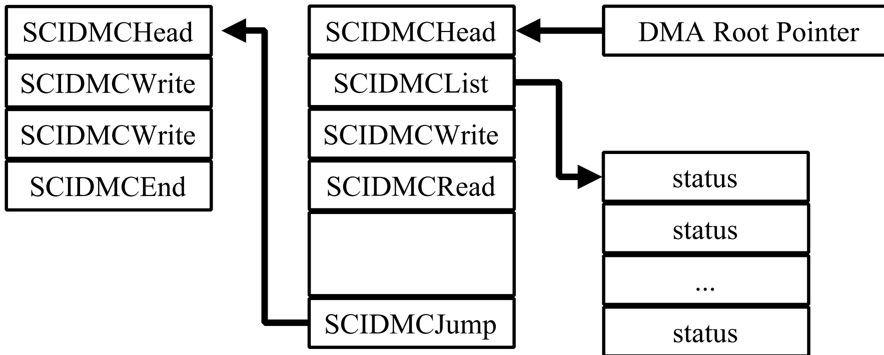


Fig. 10.6. A sketch of the chain-mode DMA descriptor architecture

descriptor list of the block move procedure into the native chain command format by generating an appropriate internal chain list. A hashing method is implemented avoiding the same descriptor chain being translated multiple times should it be used several times.

All DMA block move transactions are posted using the call-back mechanism for asynchronous synchronization. This is best applied in the typical hardware implementation, utilizing a DMA controller. If synchronous block move functionality is required, it can be implemented using the call-back procedure and semaphores supported by the given operating system. Another scenario featuring minimal latency is the use of the appropriate status list entry as a mailbox flag, which is polled by the CPU and set by a specific chain DMA command.

10.9 Message Passing Transactions

As already discussed in Section 10.5.2, a standard message passing API is required to implement higher level protocols. The IEEE Control and Status Register (CSR) Architecture standard [2] defines two standard CSR addresses for message passing, one for request messages and one for response messages. The separation of request and response messages is necessary in order to avoid possible deadlocks.

The optional, defined messaging protocol requires two message queues to be implemented in hardware or software. All required flow control preventing the loss of messages must be handled by the SCI interface and the API. Two FIFOs, which can store a minimum of 64 bytes, are required for the request and response queues in the local address space. If they are exported into the SCI address spaces accordingly, all SCI write transactions will write to these FIFOs. The FIFO full status can produce an appropriate retry message to the SCI requester.

A defined message is transmitted to a target by sending it to a specified sub-address (refer to [2]) of the SCI target node. The message is then routed to the appropriate process by virtue of its message port. The receiving node must be listening to a message with the defined port. This is done by executing a transaction specifying the message port and the appropriate exception handler, which is called every time an appropriate message is received. This message handler then receives the message as argument.

A message is a fixed length data block of type request, response or interrupt, indicated by the type bits (T), as sketched in Figure 10.7. Three fixed length sizes (16, 64 and 256 bytes) are supported and defined by the bit field labeled S. The message port is a 28-bit number that is segmented into 4 segments (port A - port D), similar to an Internet number. The following 16 bits define the sender's SCI node ID, which is required to be able to return messages to the sender. The remaining part of the message structure is the payload (b[0] - b[57]).

2	2	4	8	8	8
T	S	portA	portB	portC	portD
SID MSB			SID LSB	b[0]	b[1]
b[2]			b[3]	b[4]	b[5]
...			...	...	...
b[54]			b[55]	b[56]	b[57]

Fig. 10.7. The layout of a standard 64-byte SCI message

The IEEE Control and Status Register (CSR) Architecture standard [2] also defines a CSR register that causes a local interrupt to fire if it is written to. A specific message structure is defined similar to the one sketched in Figure 10.7, which is generated for every interrupt fired. User processes may listen to such interrupt messages similar to listening to a specific message port.

10.10 Cache Transactions

SCI networks may be cache coherent. Whether or not a given transaction is cacheable is determined by the window attributes of the appropriate address. The cache control functionality is optional. In the case of a non-coherent implementation, the procedures return signaling that this feature is not supported.

If a given memory region is used for message passing, it is necessary to be able to flush the related caches in order to ensure that a given message is actually visible to the remote nodes. Therefore, it is necessary to provide

at least some cache control functionality. The SCI Physical Layer API provides a set of procedures that provide the SCI cache control functionality, allowing one to load, flush, purge, cleanse, and delete a cache line. All cache transactions are based on a local address which identifies the cache line.

There is some need for cache control even in the case of strictly non-coherent implementations because also non-coherent hardware typically implements speculative read ahead and write posting. Whether or not speculative read ahead is allowed is determined by the cache enable bits. For example, only if the given address window is marked cacheable, may speculative read ahead be executed. In order to avoid unintended side effects when accessing CSR or other I/O registers, such address windows have to be defined non-cacheable. In the case of enabled posted writes it is necessary to be able to ensure the posted write buffers are empty. This is implemented using the cache flush procedure, which has to be provided in any case.

10.11 Conclusions

The software standard described here is a zero transaction overhead, general purpose distributed shared memory API that was designed for SCI. It is described best as general purpose shared memory hardware abstraction layer and is ideally suited as core API for higher level software architectures such as MPI [6] or VI Architecture [7]. However, it is also useful as a native layer because it implements all functionality necessary to set up and run a distributed shared memory system.

References

1. J. W. Harris and the STAR Collaboration. The STAR Experiment at the Relativistic Heavy Ion Collider. *Nucl. Phys.*, A566, 277c, 1994.
2. IEEE Standard 1212-1991. *IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses (ANSI).* ISBN 1-55937-352-0. 1993.
3. IEEE Standard 1596-1992. *Standard for Scalable Coherent Interface (ANSI/IEEE).* ISBN 1-55937-222-2. 1993.
4. IEEE Standard 1596.5-1993. *Standard for Shared-Data Formats Optimized for Scalable Coherent Interface Processors (ANSI/IEEE).* ISBN 1-55937-354-7. 1994.
5. IEEE Standard Working Group P1285. *IEEE Standard Proposal for Scalable Storage Interface.* This standard is not yet published. The latest draft is available at the working group's homepage: `http://www.SCIzzL.com/P1285/index.html`.
6. The Message Passing Interface (MPI) standard. This standard is maintained by the MPI Forum and is available on the World Wide Web: `http://www.mcs.anl.gov/mpi/index.html`.
7. Virtual Interface Architecture. This draft standard is maintained by the VI Architecture Forum and is available on the World Wide Web: `http://www.viarch.org`.

Part V

Message Passing Libraries

While the SCI services can be used directly by application programmers, higher-level message passing libraries provide a more suitable application programming interface for portable codes. This part presents projects on the design and implementation of BSD sockets, TCP/IP and the high-level message passing environments PVM and MPI—all of them optimally exploiting the fast SCI communication network.

Chapter 11 describes the SCI socket library *SSLib* developed at Technische Universität München (TUM). Compatibility to existing standards and maximum performance were the two major aims of this project. As a result, the SSLib provides exactly the application programming interface as the popular Berkeley (BSD) sockets, but based on the distributed shared memory mechanisms of SCI. Rather than building on the TCP/IP code in kernel space, the TUM researchers have re-implemented the data transfer protocol to run in user space. This results in a much improved communication latency.

Another approach has been taken by the Paderborn project team which reports on a TCP/IP implementation for Linux in Chapter 12. Instead of replacing TCP by a user-level transport protocol, their *Scalable Coherent IP (SCIP)* has been implemented as a Linux device driver. Because only the IP protocol has been adapted, few lines of code needed to be rewritten.

The *PVM-SCI* package in Chapter 13 is a lean implementation as well. It builds on the latest PVM release 3.4, which has been extended by some software to make use of the DSM capabilities of SCI when sending messages with the "PvmRouteDirect" option. When this option is not set, the standard PVM communication (via the two PVM daemons) is used. PVM-SCI is fully compatible to the standard PVM. Existing legacy applications just need to be re-compiled to make use of PVM-SCI. As a special feature, PVM-SCI automatically downgrades to other interconnects (like Fast Ethernet or ATM) when SCI is not available. This is useful in heterogeneous clusters and when communicating with front-end machines outside the cluster.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 207-208, 1999
© Springer-Verlag Berlin Heidelberg 1999

Compared to the more research-oriented PVM-SCI software, the *ScaMPI* library described in Chapter 14 is a full fledged commercial MPI product. Up to date, ScaMPI is probably the most efficient MPI implementation on SCI. According to the software developers at Scali A.S., ScaMPI is tuned towards providing high communication bandwidth and low latency, and it is flexible in the choice of the transport medium. Moreover, ScaMPI is thread-safe, which is necessary on multi-processor nodes when different threads constituting a single MPI process request services from the MPI library.

11. SCI Sockets Library

Hermann Hellwagner[1], Josef Weidendorfer[2]

[1] Institute of Information Technology, University of Klagenfurt
A–9020 Klagenfurt, Austria
email: hermann.hellwagner@uni-klu.ac.at
http://www.itec.uni-klu.ac.at/
[2] Institut für Informatik, Technische Universität München
D–80290 München, Germany
email: josef.weidendorfer@in.tum.de
http://wwwbode.in.tum.de/

11.1 Introduction

11.1.1 Rationale

Sockets [9] have become a widespread programming interface for distributed computing. A wealth of legacy applications and higher-level communication libraries relies on this API. Sockets also provide the communication infrastructure for parallel processing systems, e.g., MPI and PVM, in workstation cluster environments. An obvious way to support those applications on a compute cluster with a new-generation, high-speed interconnect such as SCI, is therefore to port the sockets API onto this platform.

In commodity networks like Fast Ethernet or ATM, the standard data transfer protocols underlying the sockets API are TCP/IP and UDP/IP. These are operating-system (OS) level, heavy-weight protocol stacks designed for wide-area communication over unreliable networks. They usually provide good sustained throughput, but fail to achieve low communication latencies.

For cluster-based parallel computing, which employs local area or system area networks, potentially with reliability supported in hardware, and where low latency is of great importance [17], the TCP/IP suite of protocols provides insufficient performance. As analyzed in depth in [12], this is due to the high per-packet processing overheads, incurred primarily by the multi-layer protocol organization, different data abstractions, layer transitions, copying costs, complicated memory management (`mbufs`), and wide area-specific processing requirements such as in-packet checksums for end-to-end reliability, computed by the communication software.

Communication systems for recent high-speed cluster interconnects therefore avoid such overheads as far as possible. Many such systems provide a new API with programming abstractions similar to the underlying interconnect hardware in order to realize highest possible communication performance. Examples are Active Messages [16], native U-Net communication [18], Illinois Fast Messages [11], and the Universal Message Passing library for the Digital Memory Channel [5][8]. The Virtual Interface (VI) Architecture [3]

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 209-229, 1999
© Springer-Verlag Berlin Heidelberg 1999

has recently emerged as a standard for user-level cluster communication; the proposed API is not part of the VI Architecture specification, though.

In order to maintain compatibility with existing applications, implementations of a standard API like sockets must adopt the alternative solution of dealing with the problem of high processing overhead [12]: changing the communication protocol and its implementation so that the native properties and communication mechanisms of the network are being exploited for improved performance. Such "fast sockets" implementations have been developed, e.g., for Myrinet NOWs [12][11][19], the SHRIMP multicomputer [1], U-Net-based networks [18], for a VI Architecture-based system area network [15], as well as for SCI compute clusters [13][14].

The software system described in this chapter, the *SCI Sockets Library (SSLib)*, also falls into the latter category. SSLib is a user-level library exporting the full Berkeley (BSD) sockets API [9], but internally making direct use of the low-latency communication mechanisms of SCI, most importantly the distributed shared memory (DSM).

The design and implementation of SCI Sockets was guided by two principal requirements: compatibility and performance. Compatibility denotes our goal to provide full support for applications based on standard, OS-level sockets. That is, SSLib should be able to become the basic infrastructure for parallel processing software (e.g., MPI and PVM) and legacy network applications using sockets, and thus open up SCI clusters for a wide base of parallel and distributed software systems. The second goal was to deliver as much of the raw SCI communication performance as possible to the applications. Therefore, rather than to reuse and adapt TCP/IP in kernel space, as described, e.g., in chapter 12, we decided to rewrite the data transfer protocols and run them in user space, such that the native SCI features are fully exploited for high-performance communication.

11.1.2 Overview

The description of the SCI Sockets communication layer in the remainder of this chapter is organized as follows. In Section 11.2, the features and the design of the SCI Sockets system are presented. Section 11.3 deals with some implementation issues. Functional tests and performance results are given in Section 11.4. Related work is addressed in Section 11.5 and conclusions are given in Section 11.6. The SSLib system is fully described in [20].

11.2 Features and Design

11.2.1 Features

The SCI Sockets system has been designed to conform to the semantics of the standard Berkeley sockets API as closely as possible. Both *stream* sockets and

datagram sockets are supported. SCI sockets and regular sockets can coexist, with SCI sockets being created by default. Currently, SCI sockets are used for communication *within* the SCI cluster only. In other words, there is no routing functionality from SCI to the TCP/IP world (or vice versa).

The SCI Sockets software provides wrappers to all Unix system calls referring to BSD sockets. For example, the SSLib `write` function makes a table lookup to detect if an SCI socket is concerned; if not, the original `write` is called. Currently, this is done by including a special header file; C macros are used for renaming. A `write` call in the application is translated to a call to `ss_write`. Although there are many ways to achieve the desired effect, we chose this method as it apparently is the most portable solution. Programs need to be recompiled, however, to be able to take advantage of SCI sockets.

Notable SSLib features—that are not supported by most other user-level socket implementations—include:

- special handling of `fork` and `exec` system calls at user level, in order to facilitate, e.g., secure sharing of SCI sockets and access synchronization between processes;
- support of the `select` call, such that the call waits on SCI sockets and kernel file descriptors simultaneously;
- support of C `stdio` runtime functions, in case a networking application makes use of them to refer to sockets;
- support of out-of-band (OOB) data known from TCP;
- asynchronous I/O;
- transparent switching among `load`/`store`-based and messaging-based data transfer mechanisms, whichever is more efficient. (These modes are called *PIO* (Programmed I/O) and *DMA* (Direct Memory Access), respectively, in Chapter 3.)

The SCI Sockets software currently runs on Solaris, Linux and Reliant Unix platforms using Dolphin SBus-SCI and PCI-SCI adapter cards (see Chapter 3). The software is designed to be portable as far as possible: the platform-dependent functionality (specific implementation of the DSM and other communication mechanisms) is hidden in a low-level communication layer exporting a well-defined API to platform-independent, higher-level functions (cf. Section 11.3).

11.2.2 Components

The functions provided by the sockets API are usually invoked through direct calls to the underlying OS, entailing expensive context switches between user and kernel spaces. For low-overhead and low-latency communication, such context switches have to be avoided in time-critical communication paths. Therefore, in the SCI Sockets system, as much communication functionality as possible is provided in a library linked to the application. This library proper is called *SCI Sockets Library (SSLib)* and is responsible for the actual

data transfers over SCI sockets, mostly via `read`s and `write`s. Such system calls are intercepted by the library if they address SCI sockets or if they may have an impact on the further use of SCI sockets. An example of the latter case occurs when a process owning an SCI socket `fork`s another process so that subsequently both processes may access the socket concurrently.

Not all aspects of sockets can be handled by a library alone. Some central management is required, e.g., for port addressing or UDP simulation. Therefore, each node running an application component using SCI sockets, also runs a special daemon process called *SCI Sockets Daemon (SSD)*. The responsibilities of the SSD include:

- establishing and terminating SCI socket connections;
- maintaining status of SCI socket connections;
- managing port addresses;
- providing support for specific system calls such as `select`, `fork`, or `exec`;
- controlling sharing of, and synchronizing access to, SCI socket connections;
- buffering incoming UDP packets.

An SSD is invoked by the local SSLib to handle all those management tasks pertaining to SCI sockets and processes that use SCI sockets. We do not regard connection setup and tear-down as time critical; thus, the additional complexity and costs of this "detour" to the SSD are held to be tolerable. The benefit of this scheme is that the SSLib and SSD can cooperatively implement the full sockets functionality usually realized by the OS.

11.2.3 Communication via the SSLib

We first describe the low-overhead, low-latency data transfer mechanism employed by SSLib. This mechanism exploits the remote memory access (RMA) facilities of SCI DSM and, more specifically, the efficient remote write capabilities of the Dolphin SCI adapter cards: buffering, combining, and streaming (see Chapter 3). The term *buffered writes* is used in the following to refer to these capabilities. Many message-passing libraries based on the Dolphin SCI devices, among them those described in other chapters of this book, utilize the same scheme to transfer data.

The data transfer is based on a shared data structure called the *Receive Ring Buffer (RRB)* in the sequel. The RRB is a ring structure where the sending (writing) node writes data into and the receiving (reading) node searches for, and copies out, data. The RRB physically resides in the local memory of the receiving node, made up of pages locked in memory; the RRB is exported into the SCI DSM, and imported and mapped into the address space of the sending process. This is depicted for bidirectional communication between nodes in Figure 11.1. Both RRBs are located in user space so that the OS need not be involved in transmitting data.

An RRB and an associated address mapping are set up for each sender and receiver pair. A bidirectional SCI socket connection thus consists of two

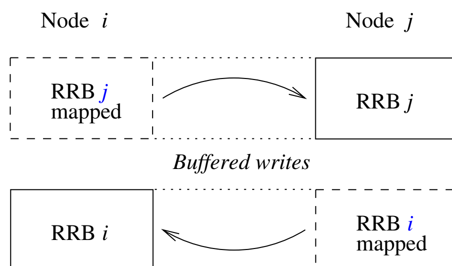


Fig. 11.1. Communication areas between nodes in SCI DSM

mutually mapped RRBs as shown in the figure. The setup procedure comprises three steps for each direction: creation of an SCI DSM segment by the receiver to host the RRB; export of the shared segment into the sending node's PCI address space; and mapping of the RRB into the virtual address spaces of the sending and receiving processes. As described below, this setup is performed on demand, when SCI sockets are established by the involved SSDs.

Once established, an RRB is maintained and used (for unidirectional data transfers) as follows (see Figure 11.2). A *start pointer* and an *end pointer*, residing in user space as well and shared among the sender and the receiver, delimit the RRB area that contains valid (sent or written) data. Before writing data, the sender checks these pointers and blocks in case the RRB is full, i.e., if an RRB overflow would occur as a result of writing the data block. Similarly, the receiver checks the pointers before copying data out and blocks if the RRB is empty. This implements a simple flow control mechanism.

After having written data into the RRB, the sender updates the end pointer; after having read data out of the RRB, the receiver updates the start pointer. As shown in Figure 11.2, the end pointer physically resides on the receiving node, with a local copy being maintained by the sender. Vice versa, the start pointer is physically located in the sender's memory, with a local copy at the receiver. This choice of the physical locations of the pointers allows them to be updated with buffered writes, while their local copies allow local read operations to be used (i.e., inefficient remote reads to be avoided) when the pointers need to be checked before data transfers. Using this scheme, only remote write operations at user level are involved in the actual data transfer over SCI sockets.

Transmitting data into and out of the RRB as shown above, involves one extra copy operation. Clearly, a user-level communication layer like SSLib, which aims at very low latencies, should avoid copying whenever possible. Unfortunately, the semantics of the original BSD sockets interface requires a copy operation to be performed, which transfers the data from a buffer in the sender's address space to the actual send buffer, and vice versa at the

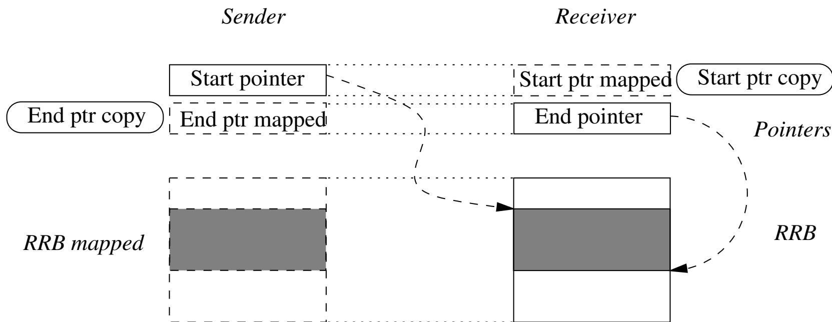


Fig. 11.2. Maintenance and use of RRBs (unidirectional case)

receiver's side. Originally, with TCP/IP, this intermediate buffer is in kernel space; in SSLib, this buffer is the RRB.

The only opportunity to avoid the extra copy is the situation when the receiver blocks in a receive call issued prior to the arrival of the data; this is called *receive posting* in [12]. In this case, the user-supplied, well-known receive buffer would have to be mapped into the sender's address space (via the SCI DSM) to enable direct data delivery, meaning that regular virtual memory would have to be treated like exported SCI memory. Apart from considerably complicating the design and implementation of SCI Sockets, this would require OS support on each receive call, e.g., for locking down the receive buffer's pages in memory, which contradicts low latencies. Therefore, we chose to always copy data via the RRB.

11.2.4 Connection Setup

Connections are set up by the SSDs. We have to distinguish the TCP-like case and the UDP-like case, i.e., stream and datagram sockets, respectively.

A TCP-like connection results from applications on two nodes calling the functions `listen`/`accept` and `connect`, respectively. As illustrated in Figure 11.3, the SSLibs involved notify the SSDs on their nodes which in turn contact each other by means of an initially established DSM segment mapping. Each SSD creates and exports a DSM segment (DSMS) to host the RRB for data reception. After having exchanged some information (e.g., the IDs of the created DSMSs and the port numbers), the SSDs hand over the information to their respective applications (SSLibs). These mutually map the DSMSs, at which point the data transfer can begin. As indicated in the figure, early `read`s or `write`s are postponed until after this point. Conforming to the BSD sockets semantics, the `connect` call returns as soon as the connect request has been delivered to the communication partner. It must be noted that the entire process is solely based on communication over SCI. No auxiliary TCP connection (established over Ethernet, for instance) is required; see Section 11.3.1.

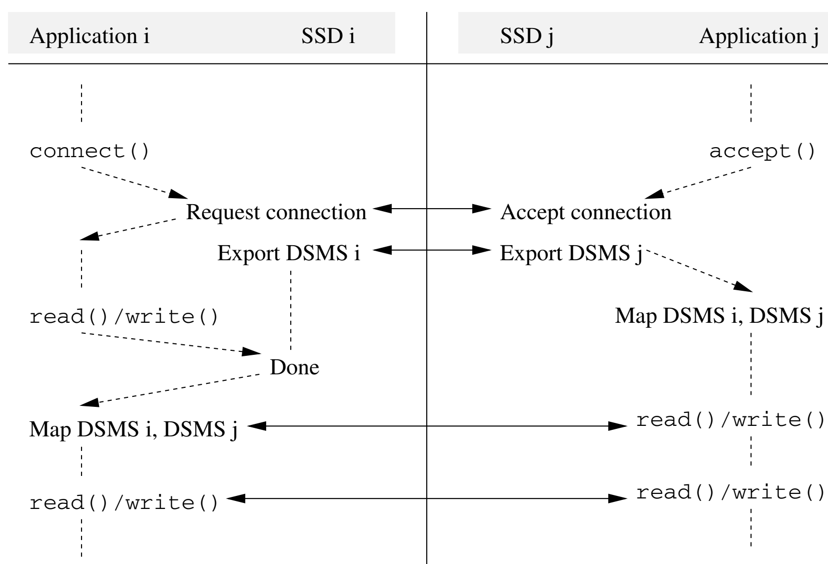


Fig. 11.3. TCP connection setup

Datagram sockets, i.e., UDP-like communications, need special treatment and pose some difficulties in SCI networks. With datagram sockets, there is no dedicated point-to-point connection and no explicit connection establishment step. Data can be sent without the need to know the receiver in advance. Furthermore, many processes can retrieve data from one endpoint, specified by a UDP port. This is in conflict to our SCI DSM-based communication model which relies on mutually mapped, dedicated DSMSs (hosting the RRBs) for data delivery, as described earlier.

The solution is that, for a UDP connection, an SSD initially plays the role of the receiving process, supplying receive buffer space (an RRB) and buffering incoming packets as needed. If an application later requests available UDP data, the ID of the DSMSs for the corresponding UDP connection is handed over to this application. All further data transfers referring to the same UDP port are treated according to the TCP case. In other words, UDP data are sent over a temporary, dedicated (TCP-like) connection. Other applications requesting data from the same UDP port (and sent by the same remote application) do not receive anything. Since datagram sockets are unreliable anyway, this is not considered a semantics violation. A fixed amount of SCI resources is used for datagram sockets. If a new UDP connection is requested and all the resources have been consumed, the SSD tears down the oldest UDP connection in order to build the new one, possibly throwing away buffered data. Again, due to the unreliability of datagram sockets, this behavior is considered acceptable.

11.2.5 Handling Special System Calls

System calls pertaining to sockets directly or indirectly, have to be intercepted and treated by the SCI Sockets software. One of the most complex system calls of this type is the `select` call. In our case, this function has to deal with regular file descriptors, with OS-level sockets, and with SCI sockets. The descriptors have to be monitored in order to detect which ones are capable of doing input or output, or which ones have exceptional conditions to report, e.g., out-of-band data. The calling process is possibly being put to sleep if none of the descriptors is ready.

As long as only SCI sockets are concerned by a `select`, the call can be handled within the SCI Sockets software by a call to an appropriate low-level wait function. If in contrast a process simultaneously waits on an OS-level condition and an SCI condition, the `select` supplied by the OS has to be called by the SSLib. Once a requested SCI condition arrives during the `select` system call, a signal has to be generated to interrupt the call. This is either effected by the low-level part of the SSLib when data arrives on an SCI socket, or by the SSD when, e.g., a connect request arrives on a listening socket or connection establishment has completed successfully. The conditions requiring asynchronous signaling are reported to the SSD by the SSLib prior to the `select` call.

Some system calls do not directly operate on SCI socket descriptors, yet have to be intercepted because they affect the further use of SCI sockets. The most important functions of this type are `fork` and `exec`.

If a process calls `fork`, it is duplicated; open file descriptors are valid both in the parent and child processes. This semantics has to hold for SCI sockets as well, so SCI socket resources have to be duplicated during `fork`. As a consequence, data structures (the RRBs) become shared between the two processes; access to them has to be synchronized subsequently. Furthermore, administrative data maintained for a connection, e.g., start and end pointers, have to stay consistent between the two processes. These two issues are solved by means of the SSD as follows. Each SCI socket has an owner which, in the regular case, is the application process having exclusive access to the socket. In case of a `fork`, the SSLib of the owner process intercepts the call, marks its SCI sockets as shared, hands over their ownership and resources to the SSD, and issues the `fork` system call. If subsequently one of the participating processes accesses an SCI socket, it has to request the ownership from the SSD before being permitted to proceed. The owning process receives and returns up-to-date and consistent values of the shared data and pointers. The SSD employs a semaphore to synchronize accesses to a shared SCI socket, potentially blocking concurrent accesses by other processes. A user process retrieves exclusive access to an SCI socket only after all other peer processes have closed the socket. This behavior is illustrated in Figure 11.4.

Clearly, due to the synchronization, data transfers over shared SCI sockets are much less efficient than over non-shared ones. Yet, this approach repre-

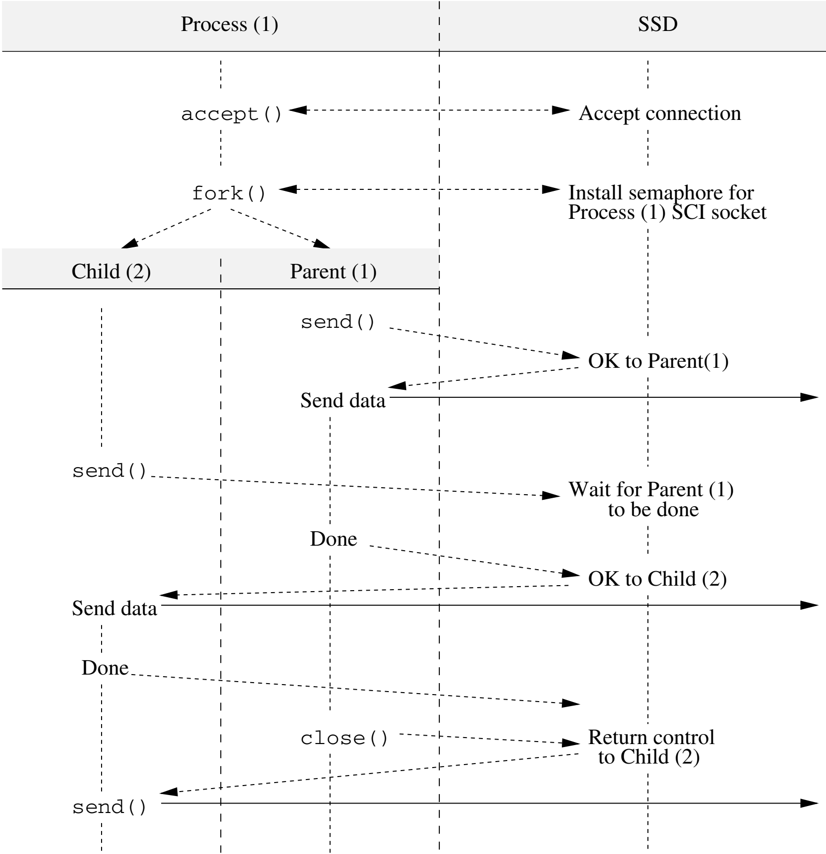


Fig. 11.4. Forking a process and sharing a connection

sents a viable solution to the problem of sharing user-level sockets which most other “fast sockets” systems do not address at all.

On calling `exec`, a process replaces its current memory image by that of a specified program. Without any modifications, this step would discard the SCI socket resources of the original process. An SSLib therefore has to make sure that its SCI sockets can be restored after an `exec` call. To that end, it intercepts the `exec` call and forwards all SCI sockets information to the SSD before calling the OS-level `exec`. Every program linked to the SSLib first registers with the local SSD. When such a program is executed after an `exec`, the SSD detects that it has already a process registered with the same process ID. It hands over any information on SCI sockets that it received from the old process. The new application can then restore the SCI sockets opened by the original process.

11.2.6 Other Calls Intercepted and Handled by the SSLib

A program can do I/O directly via system calls or via the buffering routines supplied with the C `stdio` runtime library. In addition to specific socket calls like `connect` or `accept`, the low-level I/O routines like `read`, `write`, and `ioctl` have to be supported by the SSLib. Furthermore, the C runtime I/O routines and macros, e.g., `printf`, `fgetc`, or `putc`, must be intercepted and handled by the SSLib. The reason is that some applications, e.g., the `ftpd`, make use of these routines for communication over sockets. Intercepting the latter calls incurs significant overhead for regular programs even if they do not use SCI sockets. Therefore, this feature is provided as an option only.

11.2.7 Out-of-Band Data

Regular TCP supports out-of-band (OOB) data. The SCI Sockets software does so as well, by providing a separate ring buffer at each communication endpoint for the OOB data. Thus, OOB data can be sent even if the regular communication buffer is full. On the arrival of OOB data, the receiving process is notified by a signal (if requested). Similar to TCP, a hint is sent in the regular data stream to indicate the presence of OOB data. OOB data can be read inline; hints to already read OOB data are skipped. The presence of OOB data is a condition which can be waited on in `select`.

11.3 Implementation Aspects

11.3.1 Communication Among Components

Communication among SSDs on different nodes utilizes special DSM segments (with well-known segment numbers) that the SSDs export and mutually map during startup. Each SSD has its own slot in the segment of the

other SSDs so that the SSDs can initially contact each other concurrently. An SSD's slot address (offset) is specified by the SCI ID of the adapter card of the node hosting the SSD. For the rest of the communication among SSDs, e.g., for connection setup, DSMS regions to be used are negotiated. This data structure and procedure obviates the need to resort to standard TCP- or UDP-based communication for SCI connection setup.

Communication among SSLibs and their local SSDs utilizes two forms of interprocess communication (IPC). Regular communication, which involves non-time critical messages only, is based on Unix System V Message Queues. These ensure that all messages are synchronized by the OS kernel. If however some amount of data has to be shared, Unix System V Shared Memory is used. This is, for instance, employed by an SSLib to report the conditions it wants to be signaled upon (e.g., in a `select`) to the local SSD, and by the SSD to deliver these conditions to an SSLib.

Finally, as already mentioned, Unix-style signals are used to notify other SCI Sockets software components of important events. For example, the SSD raises a signal on successful establishment of a connection, provided the user application is in a `select` call, or if UDP data is received and an application waits on data on a specified port.

11.3.2 SSLib Layers

As indicated earlier, the SCI Sockets Library is divided into two layers separated by an interface called *SCI Sockets Low-Level API (SSLLAPI)*. The upper layer comprises the high-level functions of the SSLib, like support of the sockets API, communication via DSM segments as described above, and communication between SSLib and SSD. The bottom layer isolates this functionality from the specific hardware and OS platform, e.g., the DSM implementation and SCI hardware and device driver details, making the library portable to a large extent.

The functionality and mechanisms supported by the bottom SSLib layer can be divided into four groups:

- *Basic SCI DSM management mechanisms.* These functions provide for creating, exporting, importing, mapping, unmapping, and destroying of shared memory segments required for data transfers. It must be noted that the upper layer need not be concerned about the type of shared memory being used. In the regular case, the shared memory will be comprised of SCI DSM segments, but it may also be a mapped, shared file or local shared memory. Using the latter implementations, the SSLib could be readily tested and debugged on a single node and a shared-memory multiprocessor, respectively.
- *Control transfers.* In addition to data transfers, control transfer is required in a communication library like the SSLib. For this purpose, the bottom

SSLib layer supports a *"notifying" write* which notifies a process of a modification to a DSM location. Such writes are used, for instance, to update RRB start and end pointers. The SSLLAPI functionality for control transfers comprises:
  - a function actually performing a notifying write operation;
  - a function allowing to wait for, and synchronously react on, a modification to one of the specified DSM locations, until a timeout expires;
  - registration of a call-back function that is to be asynchronously invoked on a notifying write to a specified DSM address;
  - temporary disallowing and postponing of call-backs in case a process executes a critical region, for instance.
- *Memory barriers.* A serious drawback of SCI is that it does not guarantee in-order delivery of transactions. Thus, it would, e.g., be possible for data written into an RRB to arrive after the corresponding RRB end pointer update becomes effective, potentially leading to erroneous data being read at the receiving end of the connection. The solution implemented in the lower SSLib layer is that notifying writes guarantee memory ordering as well. That is, a notifying write ensures that all outstanding data transfers (writes) have completed before it is being performed. This is accomplished by means of the *store barriers* supplied by recent versions of the Dolphin SCI adapters, or simply by using remote DSM read operations.
- *Management of DMA-based channels over SCI.* Most Dolphin SCI adapter cards offer two mechanisms for data transfers over SCI: (1) the processor actively copying data via DSM segments, as introduced above, and (2) "background" DMA transfers, involving the processor only in the start-up phase. For the latter data transfer mechanism, a group of functions in the SSLLAPI supports opening and closing unidirectional read or write DMA channels between the communication partners; DMA transfers offered by the SCI device driver can then be used.

11.3.3 Choice of Most Efficient Communication Mechanism

Since the DMA-based data transfers require device driver (OS) intervention to trigger the DMA engines on the SCI adapters, they entail context switch overhead and increased latency. Thus, DMA is most useful for moving large amounts of data, which amortizes start-up costs and relieves the processor from the time-consuming copying task. The SSLib transparently selects the most efficient communication mechanism for a given data transfer, depending on the data volume to be sent. The threshold that switches between the transfer options, e.g., 4 kByte size, can be compiled in or specified as a runtime parameter by the user.

Unfortunately, some versions of the Dolphin SCI cards show poor performance on DMA transfers; achievable peak bandwidth is lower than using the copy mechanism. In these cases, the SSLib defaults to copying as a result of a high threshold value.

11.3.4 SSLib Implementations

The SCI Sockets software was initially developed on Solaris platforms (SPARC-stations-2 and UltraSPARCs with Dolphin SBus-1 and SBus-2 SCI adapter cards, respectively) and later ported to Linux systems (Pentium II-based PCs with Dolphin's SCI-PCI adapters, Rev. B) as well as to Siemens Nixdorf RM x00 workstations with the Reliant Unix OS and SCI-PCI cards as well. Porting did not pose problems.

For testing purposes, the initial versions were restricted to run on a single node, employing shared, mapped files for shared memory and the `kill` system call to implement control transfer: a Unix signal is asynchronously delivered to the receiving process as a result of a notifying write, or can be synchronously waited upon. The SSLib implementation on Solaris was then changed to utilize an extra thread to poll for notifying writes, as explained below. Further, real SCI DSM segments were supported, and this cluster implementation was ported to Linux and the Dolphin SCI-PCI adapters. Finally, support for the remote interrupts supplied by these SCI cards was added.

11.3.5 Control Transfers

A crucial issue in user-level communication systems is the implementation of control transfers, most importantly notifying the receiver of incoming messages. Two models exist: interrupt-driven and polling-based message handling. Because of its importance, we describe and justify the control transfer approach in the SCI Sockets software, which is mainly based on polling.

For the first implementations of the SSLib, there was no support for a remote interrupt-driven notification mechanism. Thus, polling of specified DSM locations was used, to detect modifications when needed. On Solaris, for example, an extra kernel thread is created by the bottom layer of the SSLib, responsible for polling the DSM locations which are subject to notifying writes. The two cases of control transfer are handled as follows.

In the first case, i.e., if call-back handlers are registered and the user application does some computation or blocks in a system call, the application has to be interrupted. The extra thread actively polls the DSM locations for which call-back handlers are registered. On a modification, this thread sends a signal to the main application thread. In this manner, system calls like a `select` are interrupted. A Unix handler for the signal is called which searches for the change and transfers control to the according handler function(s).

In the second case, that is, on a call to the wait function supplied by the SSLLAPI, the main thread suspends the polling thread and itself polls the addresses specified in the wait function. On a change, the corresponding handler is called or the function returns, starting the polling thread again.

Thus, the second thread is active only if there are handlers registered and the wait function has not been invoked. In the SSLib, this happens mainly when asynchronous I/O is switched on for an SCI socket, or when the main

application thread is blocked in a `select`, which had to call the OS-level `select` and is simultaneously waiting for a condition on an SCI socket.

In the first case, latency is far from good. The scheme is supplied for asynchronous notifications, for which latency is in many situations not of primary importance: the application is busy doing some work anyway, overlapping computation and communication. The second one is the case where low latency is important, i.e., when the application is blocked waiting for communication. The polling approach provides low latencies in this case.

A recent investigation [7] quantified the performance of handling messages via interrupts and polling, concluding that a hybrid system achieves robust performance in most of the situations. It relies on polling during application idle times and uses interrupts otherwise. Since recent Dolphin SCI adapters provide support for raising remote interrupts, such a hybrid scheme for the SSLib became possible in principle. Our experiences with remote interrupts over SCI are, however, that their latencies are too high (some 100 μs) to render them feasible as the sole notification method, and that enabling and disabling interrupts from user level on the receiver side is too difficult to flexibly switch between the two mechanisms in a hybrid system.

11.4 Functional Tests and Performance

Common Unix tools that regularly use standard BSD sockets, e.g., `ftp`, `ftpd`, and `inetd`, have been used to test the correct behavior of SSLib. The programs run well and deliver good performance. For example, with large files `ftp` achieves nearly the throughput of the micro-benchmarks shown below. The `inetd` daemon, the standard “superserver” for Unix systems, in conjunction with the `ftpd` served to test the concepts developed for `fork` and `exec` support. Depending on a configuration file, `inetd` listens on specified network ports and starts programs on incoming connections.

Initial performance tests were conducted on an SCI cluster of eight Ultra-SPARC workstations at the University of California, Santa Barbara (UCSB). The cluster nodes are equipped with 143 MHz Ultra-1 and 167 MHz Ultra-2 processors, respectively, and Dolphin SBus-2 SCI adapters. The peak bandwidth for the raw communication primitives is reported as 30.4 MByte/s for 64-byte block stores (supported by the UltraSPARC VIS instruction set) and 25.5 MByte/s for DMA-based communication; minimum round-trip latencies are 8 μs for writes (two writes constitute the round trip) and 6 μs for reads (which is inherently a round-trip communication) [6].

Simple micro-benchmarks were used to assess throughput and round-trip latencies of the SCI Sockets software on this cluster. The results are shown in Figures 11.5 and 11.6. The performance figures are encouraging: maximum throughput of 25 MByte/s and round-trip latencies of about 17 μs for 1-byte packets. Note that this latency figure adds just a factor of about two to the

raw round-trip times, and outperforms almost all of the other "fast sockets" packages referred to in Section 11.5.

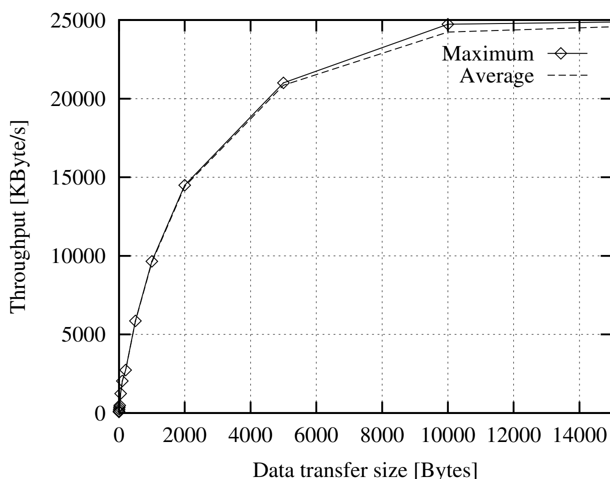


Fig. 11.5. SCI Sockets throughput on the UCSB SCI cluster

The sawtooth shape of the latency curve stems from the use of the `memcpy` function in the sender's SSLib to move data into the RRB. On the UltraSPARC processor, VIS block store instructions are transparently utilized by `memcpy` whenever possible. Note that this latency curve suggests that, for small amounts of data, local memory copy performance to a large extent determines the data transfer performance over the SCI interconnect.

The micro-benchmarks were also run on a Linux platform, comprising PC nodes with a 233 MHz Pentium II processor each, a 33 MHz, 32-bit PCI bus, 440 FX chip-set, and the Dolphin SCI-PCI adapters, revision B. Maximum throughput of 37 MByte/s and minimum round-trip times of about 25 μs have been achieved [4].

To assess the performance of TCP/IP protocol implementations, higher-level benchmarks are available as well. The SSLib library was tested with the wide-spread `ttcp` and `netperf` benchmarks. The `ttcp` program on the UCSB SCI cluster reports maximum throughput of 25 MByte/s, and a half-performance length of 1500 bytes (the size of the data transfer resulting in half of the maximum throughput). This value is higher than that of other "fast sockets" systems (typically, 500–600 bytes), reflecting the fact that `ttcp` captures the connection establishment costs as well. Connection setup is more expensive in the SCI Sockets software than in the other systems, since our system refrains from falling back to a standard Ethernet connection, as most other systems do.

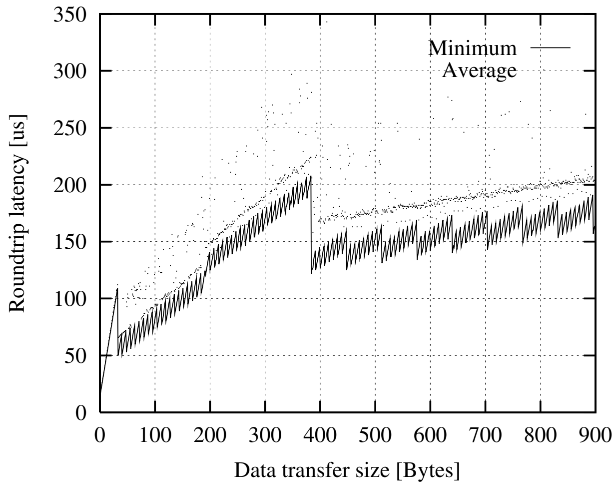


Fig. 11.6. SCI Sockets round-trip latencies on the UCSB SCI cluster

11.5 Related Work

Several user-level communication systems have been developed that support a sockets API as well.

U-Net [18] was the first communication architecture to allow protected, user-level access to the network interface on off-the-shelf hardware (SPARCstations with Fore Systems ATM adapters) running a standard operating system (SunOS). U-Net pioneered many of the ideas and concepts to be incorporated into the VI Architecture specification [3] recently. The ATM adapter firmware was modified to implement the virtualization of the network interface so that multiple user processes can communicate with the adapter concurrently. This base-level functionality was used to re-implement many TCP and UDP modules in user space, removing many of the restrictions of an OS-level implementation, e.g., redundant copy operations and complex network buffer handling. In addition, the migration to user level made the protocols more flexible for tuning to high-speed networks and adapting themselves to the state of the communication system. In contrast to the SSLib software, the base TCP and UDP protocols have not been eliminated, though.

Several user-level “fast sockets” implementations have been developed for Myrinet-based clusters [12][11][19]. All three sockets APIs have in common that they discard the standard TCP/IP protocol stack and rely on lower-level communication libraries specifically optimized for Myrinet communication.

For instance, Berkeley Fast Sockets [12] are built upon Active Messages, a reliable low-overhead transport and notification mechanism, yet not guaranteeing in-order delivery. Berkeley Fast Sockets optimize for local-area transport by omitting features of TCP/IP unnecessary in that environment. It collapses layers of the sockets API and the protocols, simplifies buffer ma-

nagement, and avoids copy operations whenever possible. Receive posting (see Section 11.2) is introduced as a means to avoid a copy through the receive socket buffer when the final destination of the data is already known upon packet arrival (through a `recv`). In contrast to SSLib, the Berkeley Fast Sockets can revert to standard protocols for non-local communication. They have several limitations, however, including lack of support for `fork` (i.e., sharing of connections), `exec`, `select`, and asynchronous I/O; connection establishment must rely upon a standard TCP/IP connection. Round-trip latencies are about 61 μs on a cluster of UltraSPARC-1 nodes connected via Myrinet, and a maximum throughput of about 33 MByte/s is achieved.

The sockets implementation based upon Illinois Fast Messages [11] and the ParaStation-2 sockets [19], which are based upon the so-called *ports* abstraction, share part of the concepts and limitations of Berkeley Fast Sockets (details are not available in the papers), with ParaStation-2 outperforming the other implementations.

SHRIMP [1] introduced the concept of Virtual Memory Mapped Communication (VMMC) which is similar to SCI DSM in that it allows applications to transfer data directly between two virtual memory address spaces over the network. Based on import-export mappings similar to SCI, two transfer strategies are supported: deliberate update, which is an explicit transfer (send) of data, and automatic update, which reflects operations on exported local memory segments in the remote memory (by hardware means). In contrast to SCI, in-order delivery of all data transfers is guaranteed and a notification mechanism is available that transfers control to a user-specified, user-level handler function after a message transfer. SHRIMP implements stream sockets only, i.e., TCP semantics. The implementation lives in user space and is based on moving data into mutually mapped RRBs, similar to the SSLib. Variants with one and two copy operations are available; the automatic-update version, which is almost identical to the SSLib data transfer, always does two copy operations in order not to violate the sockets semantics ("receive posting" is not specifically supported). Since many of the concepts are very similar to the SSLib, it is not surprising that the minimum latency is very low as well: 22 μs for a round trip. Maximum throughput is close to the raw hardware one-copy limit of about 13.5 MByte/s, over the proprietary backplane interconnect based on the Intel Mesh Routing Chip. As compared to the SCI Sockets software, the SHRIMP stream sockets library has some severe restrictions: no support for datagram sockets, no support for `fork` and `exec` calls (`select` is supported, though), and the need to fall back to regular TCP sockets for connection setup and tear-down. SHRIMP sockets were later ported to Myrinet [2].

To our knowledge, the first implementation of sockets over a VI Architecture-based network is described in [15]. This stream sockets software is based on a TCP stack migrated to user level and tailored to the VI-specific abstractions and mechanisms, with TCP features omitted where not needed.

For example, since the underlying network supports *reliable delivery* (in VI Architecture terminology), fragmentation and re-assembly of long messages can be greatly simplified in that, e.g., sequence numbers, timeouts, duplicate detection, acknowledgments, and retransmissions need not be incorporated. The system is implemented on the GigaNet cLAN GNN1000 interconnect (1.25 GBit/s one-way bandwidth) which has VI functionality implemented in hardware on the network adapter. The main contributions of this work are a credit-based flow control mechanism, user-level TCP protocol processing adapted to the VI Architecture concepts, caching of communication buffers, and reduced CPU processing overhead for communication as compared to legacy TCP implementations. Round-trip latencies of the VI stream sockets system are about 75 μs minimum (on 400 MHz Intel Pentium II Xeon-based servers), 2 to 3 times better than legacy protocols; maximum throughput is about 87 MByte/s, a factor of 3 to 4 improvement. The paper does not report about the difficult functional features of a user-level sockets implementation, like connection setup, `fork`, `exec`, or `select` support, or OOB data.

A user-level socket implementation for SCI, called *SCILAN*, is described in [13] and [14], for a Windows NT cluster with 200 MHz Pentium nodes and Dolphin PCI-SCI adapters, revision B. An interesting feature of SCI-LAN is that existing networking applications can utilize the socket library without re-compilation or relinking. This is achieved through a modified run-time linking step which redirects all socket calls from the regular WinSock library to the replacement SCILAN library on a process-by-process basis. SCILAN sockets bypass the OS and communicate directly through SCI DSM segments (mutually mapped RRBs) in user space, identical to the SSLib. SCILAN introduces a novel control transfer scheme for SCI networks: interrupt flags realized through special SCI mappings and stimulating writes. The important thing to note is that a sender always stimulates the interrupt flag in the receiver whenever it has finished a data transfer. Since the receiver can disable the interrupt flag, this may or may not result in an interrupt at the receiving end. A busy receiver will disable interrupts, whereas a receiver going to sleep while waiting for data will enable it. Selective disabling of interrupts attempts to minimize the number and costs of interrupts. A flag on the sender side allows to implement a basic flow control mechanism. SCILAN is based on the WinSock 1.1 specification, which states that socket descriptors are not valid OS-level file descriptors. This eliminates the need to intercept system calls and take appropriate actions for the user-level sockets; details on this issue are not given, though. Furthermore, SCILAN relies on SCI hardware reliability, but it remains unclear what is being done about out-of-order delivery. SCILAN achieves minimum latencies of about 16 μs when polling is used for receiver notification, and of 180 μs when interrupts are used; maximum throughput is about 21 MByte/s.

Finally, a commercial TCP/IP suite for SPARC-based SCI clusters is also available [10], through a software layer called the DLPI (data link pro-

vider interface). Regular TCP is used, with features omitted when the SCI hardware or driver provide adequate support, e.g., reliable delivery. Latency on UltraSPARC-2s with Dolphin SBus-2 SCI adapters is 199 μs, with room for optimizations pointed out by the authors; peak throughput is about 29 MByte/s.

11.6 Conclusions

This paper described the SCI Sockets Library, a user-level implementation of the legacy BSD sockets API widely used for distributed and parallel computing, for SCI-based clusters of workstations or PCs. The basic features and design issues of the system were introduced. SCI Sockets provide the functionality of standard, OS-level sockets as completely as possible, allowing legacy networking applications to be migrated to SCI clusters by simple recompilation.

The challenges that had to be overcome to achieve this include: connection establishment and shutdown at user level; interception of system calls pertaining directly or indirectly to sockets (with `fork`, `exec`, and `select` being the most difficult cases); sharing of, and synchronizing concurrent accesses to, connections at user level; support for datagram sockets; providing the out-of-band data path; and supporting the C `stdio` library as well as asynchronous I/O. Due to these features as well as its excellent performance, e.g., 17 μs minimum round-trip latencies, the SCI Sockets Library compares favorably to other "fast sockets" implementations over cluster networks reported in the literature.

Current restrictions of SCI Sockets are that it limits communication to the SCI cluster, i.e., it does not fall back to standard OS sockets for communication to non-SCI nodes; high memory consumption due to potentially large buffers locked down in memory at the receiving ends of open connections; and the predominant use of polling to realize receiver notification, in spite of situations where interrupts would be advantageous [7]. Concepts and prototypes exist to overcome these limitations in our future work, e.g., a DSM segments manager maintaining large DSM areas, portions of which are allocated to SSLib instances on demand.

Acknowledgments

The support from Klaus E. Schauser and his research group at the University of California at Santa Barbara, for providing access to their UltraSPARC SCI cluster, is gratefully acknowledged.

References

1. S. N. Damianakis, C. Dubnicki, E. W. Felten. Stream Sockets on SHRIMP. *Proc. CANPC'97* (First Int'l. Workshop on Communication and Architectural Support for Network-based Parallel Computing), LNCS 1199, Springer Verlag 1997.
2. C. Dubnicki, A. Bilas, Y. Chen, S. N. Damianakis, K. Li. SHRIMP Project Update: Myrinet Communication. *IEEE Micro*, Jan./Feb. 1998, pages 50–51.
3. D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, March/April 1998, pages 66–76.
4. M. Eberl, H. Hellwagner, W. Karl, M. Leberecht, J. Weidendorfer. Fast Communication Libraries on an SCI Cluster. *Proc. SCI Europe '98*, ISBN 1-901864-02-2, Cheshire Henbury 1998.
5. R. Gillett, R. Kaufmann. Using the Memory Channel Network. *IEEE Micro*, Jan./Feb. 1997, pages 19–25.
6. M. Ibel, K. E. Schauser, C. J. Scheiman, M. Weis. High-Performance Cluster Computing Using SCI. *Proc. Hot Interconnects V*, Stanford Univ., Palo Alto, CA, USA, Aug. 1997.
7. K. Langendoen, J. Romein, R. Bhoedjang, H. Bal. Integrating Polling, Interrupts, and Thread Management. *Proc. Frontiers'96: 6th Symp. on Frontiers of Massively Parallel Computation.* IEEE Computer Society Press 1996.
8. J. V. Lawton, J. J. Brosnan, M. P. Doyle, S. D. O Riordain, T. G. Reddin. Building a High-performance Message-passing System for MEMORY CHANNEL Clusters. *Digital Technical Journal*, Vol. 8, No. 2, 1996, pages 96–116.
9. S. J. Leffler, M. K. McKusick, M. J. Karels, J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley 1989.
10. K. Omang, B. Parady. Performance of Low-Cost UltraSPARC Multiprocessors Connected by SCI. *Proc. CNDS'97* (Conf. on Communication Networks and Distributed Systems Modeling and Simulation). Society for Computer Simulation 1997.
11. S. Pakin, V. Karamcheti, A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency*, April-June 1997, pages 60–73.
12. S. H. Rodrigues, T. E. Anderson, D. E. Culler. High-Performance Local Area Communication With Fast Sockets. *Proc. USENIX Symposium 1997.*
13. S. J. Ryan, H. Bryhni. *SCI for Local Area Networks* Research Report 256, Department of Informatics, University of Oslo, Jan. 1998.
14. S. J. Ryan, H. Bryhni. Eliminating the Protocol Stack for Socket Based Communication in Shared Memory Interconnects. *Workshop PC-NOW'98* (First Int'l. Workshop on Personal Computer-based Networks of Workstations), held in conjunction with IPPS/SPDP'98, March 30 - April 3, 1998, Orlando, Florida, USA.
15. H. V. Shah, C. Pu, R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. *Proc. CANPC'99* (Third Int'l. Workshop on Communication and Architectural Support for Network-Based Parallel Computing), LNCS 1602, Springer Verlag 1999.
16. T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. *Proc. 19th Int'l. Symp. on Computer Architecture.* ACM Press 1992.
17. T. von Eicken, A. Basu, V. Buch. Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, Feb. 1995, pages 46–53.

18. T. von Eicken, A. Basu, V. Buch, W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *Proc. 15th ACM Symposium on Operating System Principles.* ACM Press 1995.
19. T. M. Warschko, J. M. Blum, W. F. Tichy. A Reliable Transmission Protocol for Myrinet. *Second Workshop on Cluster Computing*, March 25-26, 1999, Karlsruhe, Germany.
20. J. Weidendorfer. *Entwurf und Implementierung einer Socket-Bibliothek für ein SCI-Netzwerk.* Diploma Thesis, Institut für Informatik, Technische Universität München, Feb. 1997.

12. TCP/IP over SCI under Linux

Hüseyin Taskin, Roger Butenuth

Operating Systems and Distributed Systems Research Group,
University of Paderborn, Germany
email: {bond, butenuth}@uni-paderborn.de
http://www.uni-paderborn.de/cs/heiss/

12.1 Introduction

In this chapter, we present how SCI devices can be integrated in the TCP/IP protocol stack of Linux for high speed communication within a cluster. We developed a packet driver which is using SCI at the physical layer to enable a data transfer with high throughput and low latency. In our solution we achieved a throughput of 31 MByte/s and a latency of 77 μs. Although the size of the protocol stack implies higher software latency than hardware latency, we can benefit from the higher bandwidth compared to traditional network technologies. This makes it conceivable to replace an existing Ethernet network by our approach in SCI clusters. Instead of replacing TCP by a user level transport protocol as done by [2] and [3] (see also Chapter 11), our solution is to implement SCIP (*Scalable Coherent Interface IP*) as a packet driver, sending and receiving IP packets. This is more straightforward since SCIP, like e.g. PLIP, SLIP or Ethernet, works underneath the IP layer (see also Figure 12.1). The key advantage is that we keep the socket semantics and therefore all applications work without any modifications or recompilations. In addition, we can realize this with a small amount of source code, due to the fact that we only have to transmit an IP packet through the SCI interconnect. So SCIP is a thin layer between IP and the SCI driver interface. Another important fact is that SCIP runs without any problems on multiprocessor machines since the Linux kernel provides coordinated access to the driver.

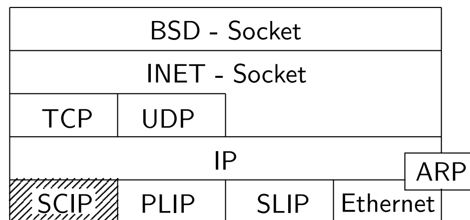


Fig. 12.1. Linux networking layers

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 231-237, 1999
© Springer-Verlag Berlin Heidelberg 1999

The chapter is organized as follows. An overview of the SCIP structure and implementation is given in Section 2. Section 3 presents some measurements of SCIP which are compared to those of other implementations. We conclude the chapter with summarizing remarks in Section 4.

12.2 SCIP Structure

12.2.1 Packet Driver Interface

Although Linux has a monolithic kernel it allows to dynamically link components of the operating system, called *modules*, without building a new kernel. The SCI device driver and the SCIP packet driver are realized as modules and can be loaded and unloaded as they are needed. There is a clear separation between the TCP protocol and the packet driver which makes it possible to hide the physical transmission from the protocol and protocol details are hidden from the packet driver. In the following we refer to a *packet interface* as an interface at the OSI level 2 which provides at least the following device methods:

– `open` and `close` methods for the interface,
– a method for the transmission of a packet,
– packet header manipulation methods,
– a method to get statistics about the interface.

In our case we need two further optional methods. One to perform interface-specific `ioctl` commands and a method to change the maximum transfer unit (MTU). The packet reception is interrupt-driven. The interface interrupts the processor to signal that a new packet has arrived and the packet must be handed over to the upper layers.

12.2.2 Hardware Address Resolution

After loading the SCIP module, we redirect the route of IP packets to use SCIP as a packet device to reach IP hosts within the SCI cluster. The following usual network configuration commands can be used:

– `ifconfig scip sourcehost`
– `route add -host desthost scip`

In order to attain a higher efficiency, we use a separate IP address translation within the cluster. The Address Resolution Protocol (ARP) commonly used by Ethernet is not usable because it needs broadcast functionality.

Our IP address translation associates IP numbers with SCI node-IDs. The table with the required data is loaded on every host in the SCI cluster by invoking the `ioctl` interface method. This is the only special program

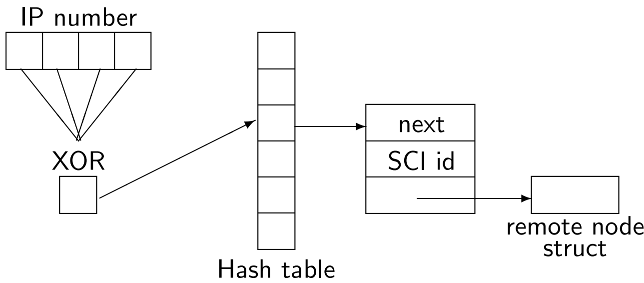


Fig. 12.2. SCIP address translation with hash table

we need to configure SCIP. The table is organized as a hash table (see also Figure 12.2). The hash function computes the XOR operation over the 4 bytes of the IP number and determines the key for the hash table entry. We assume that a usual SCI cluster has fewer than 256 nodes which corresponds to the hash table size. This simple hash function is optimal (no collisions) when all nodes are located in the same subnet with an 8-bit netmask. In case of a collision the item will be entered in a linear list (see also Figure 12.2)

12.2.3 Other Implementation Issues

Each node in the SCI interconnect provides the following resources:

- a shared memory receive buffer,
- a buffer index (next write position),
- and an interrupt flag.

The buffer is managed as a ring buffer. If a remote host reads the buffer index through the fetch and increment mapping, the buffer index will be atomically incremented. The actual index of the buffer slot where the packet will be written is the buffer index modulo the number of buffer slots. The number of buffer slots has to be a power of 2 because the fetch and increment counter wraps around after 2^{32} operations. The required number of buffer slots has been determined experimentally and an overflow of the ring buffer is extremely unlikely for two reasons. Firstly, it can be expected that the CPU copies packets faster from the local buffer than a remote sender can deposit them there. Secondly, the flow control of TCP prevents an unbound number of packets to be sent. As a result, the receiver can allow several nodes to write at the same time to the receive buffer. Instead of providing buffers for each node, we can handle this with only one ring buffer.

The size of a buffer slot corresponds to the maximum transfer unit (MTU). In our case, we chose the Ethernet MTU sizes of 1500 bytes and 15000 bytes respectively, but of course SCIP can be configured for any MTU size. The interrupt flag is a 32-bit word in the local memory. Write requests of the

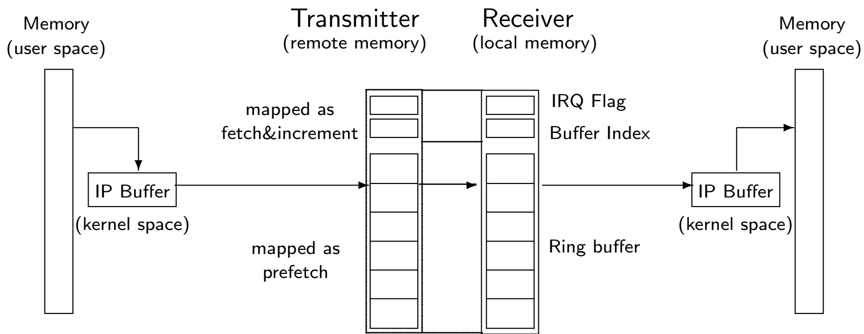


Fig. 12.3. Internal shared memory structure of two connected machines

transmitter nodes to the flag will be translated to interrupting write requests through the special mapping. It is possible to allow a number of write requests before triggering an interrupt. For this reason, the interrupt flag has to be set with an initial value and reset after each interrupt. SCIP interrupts the receiver each time it sends a packet to the receiver, i.e. the flag is set so that each write to the flag will trigger an interrupt. For the first transmission, the transmitter must map the physical memory of the receiver. After that, it reads the buffer index, writes remotely to the corresponding buffer slot and triggers the interrupt of the receiver to indicate that a new packet has arrived. On the other side, the receiver node is interrupted, copies the new entry from the buffer and hands it on to the upper IP layer (see also Figure 12.3).

12.3 Performance

12.3.1 Configuration

Performance measurements have been made with two different configurations. One consisted of two PentiumPro 200 MHz systems with Intel 440FX chip-set and 66 MHz mainbus frequency; the other of two Pentium II 400 MHz systems with Intel 440BX chip-set and 100 MHz mainbus frequency, additionally equipped with fast SDRAM. Both systems were equipped with a Dolphin PCI/SCI adapter using the LC-2 chip and PSB Revision C. Linux (Kernel 2.0.32) and Solaris were used as operating systems.

12.3.2 Latency

The latency measurements shown in Table 12.1 were made on the Pentium II system. Since both SCIP and Fast Ethernet have to move the data through the same protocol stack, there is no advantage of SCIP using SCI at the physical layer. Thus the throughput for small messages is almost equal for

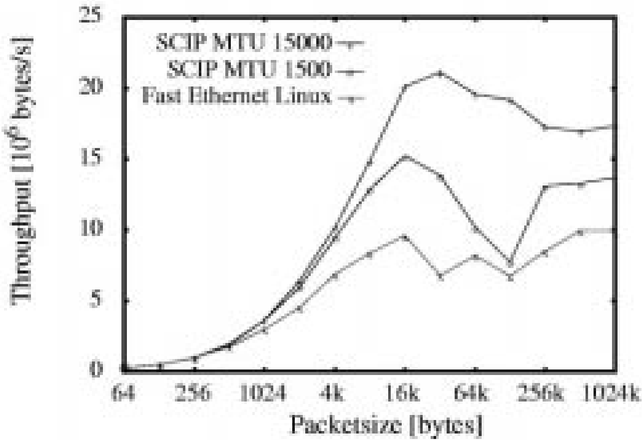


Fig. 12.4. Throughput on the PentiumPro system (440FX chip-set)

both technologies. The difference of about a factor of two between the latency figures of Fast Ethernet running under Linux and Solaris is obviously caused by the different implementations of the TCP/IP protocol. To compare our approach with the solution of [2] we made latency measurements on a Pentium 166 MHz system. The SCILAN interrupt approach of [2] has a latency of 180 μs running on the Windows NT 4.0 operating system. SCIP running on Linux has a latency of 165 μs.

Technology	**Latency** [μs]
SCIP (Linux)	77
Fast Ethernet (Linux)	73
Fast Ethernet (Solaris)	149

Table 12.1. Latency on the Pentium II system (440BX chip-set)

12.3.3 Throughput

Figures 12.4 and 12.5 show the throughput results for both configurations. The throughput of the Pentium II system is much higher than that of the PentiumPro caused by the higher memory bandwidth of the former, which allows faster local memory-to-memory copy operations. SCIP throughput outperforms Fast Ethernet when transmitting large messages in particular when an MTU size of 15000 bytes is used. The maximum throughput for the Pentium II is 31 MByte/s and 20 MByte/s on the PentiumPro, respectively. The performance drop for certain message sizes in Figure 12.4 is caused by a small bug in the TCP implementation of Linux [4]. For the Pentium II

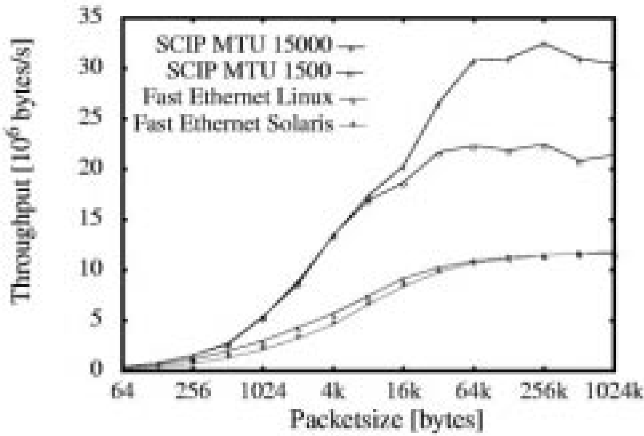


Fig. 12.5. Throughput on the Pentium II system (440BX chip-set)

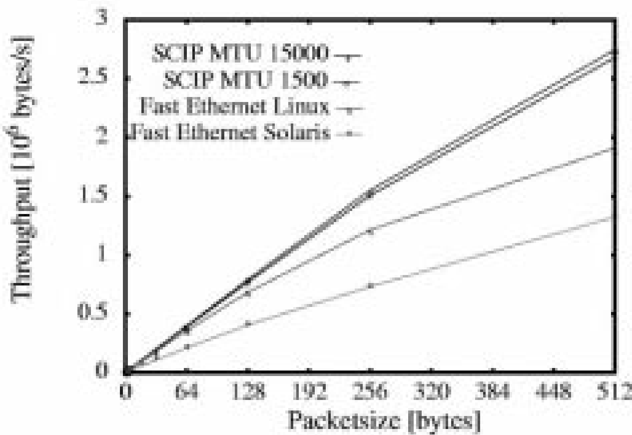


Fig. 12.6. Throughput for small messages on the Pentium II system (440BX chip-set)

system, a patch could be applied and the performance shown in Figure 12.5 could be obtained. Figure 12.6 shows the throughput for small messages from 1 to 512 bytes for SCIP under Linux and Fast Ethernet running under Linux and Solaris. Especially for small messages, there is a significant difference between Fast Ethernet under Linux and Fast Ethernet under Solaris. The smaller latency of Linux results in a better throughput for small packets. For this reason we do not expect good performance of an SCIP implementation under Solaris.

12.4 Conclusion

We have shown that using SCI at the lowest layer in the protocol stack works well and has the advantage that all existing applications work without any modifications. It is conceivable to use SCIP as a substitute for the existing Ethernet network in SCI clusters. Another important fact is that, although SCI does not provide a guarantee for correct delivery of programmed I/O operations, we can ensure that in our solution TCP/IP will correct these errors. This makes it possible to use the simple ring buffer without flow control.

References

1. A. Rubini. *Linux Device Drivers.* O'Reilly and Associates, 1998.
2. S.J. Ryan, H. Bryhni. *SCI for Local Area Networks.* University of Oslo, Research Report no. 256, January 1998.
3. J. Weidendorfer. *Entwurf und Implementierung einer Socket-Bibliothek für ein SCI-Netzwerk.* Diploma Thesis, Technische Universität München, 1997.
4. L. Prylli, B. Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In: *Workshop PC-NOW at IPPS/SPDP98*, Orlando, USA, 1998.

13. PVM for SCI Clusters

Markus Fischer[1], Alexander Reinefeld[2]

[1] Universität Mannheim, Germany
email: mfischer@mufasa.informatik.uni-mannheim.de

[2] Konrad-Zuse-Zentrum für Informationstechnik, Takustr. 7, D-14195 Berlin
email: ar@zib.de
http://www.zib.de/

13.1 Overview

PVM-SCI is a complete implementation of the PVM (parallel virtual machine) message-passing environment for SCI clusters. It provides two complementary communication mechanisms: the conventional TCP and UDP protocols that are also used by the standard PVM for sending messages from one task to another via the PVM daemons, and a special-purpose protocol that utilizes the fast SCI network when the `PvmRouteDirect` option is set. In the latter case, no time-consuming software protocol is used, resulting in a significantly reduced communication latency and improved data throughput.

With our work on PVM-SCI, we aimed at both, providing maximum communication performance over SCI and supporting transparent downgrading to other communication media (e.g. Ethernet, ATM) when SCI is not available. Hence, we have implemented PVM-SCI as a heterogeneous message-passing library that supports several interconnects and system environments.

PVM-SCI is fully compliant with the latest PVM release 3.4 It runs on Solaris, Linux and Windows NT.

13.2 Parallel Virtual Machine

The *Parallel Virtual Machine (PVM)* [4] is one of the most prominent programming environments for networked computers. Applications can be written in Fortran or C and parallelized by using PVM message-passing constructs that are mapped onto the underlying distributed system architecture. PVM supplies functions to automatically start up tasks on a distributed memory system. An authenticated PVM task can then synchronize and communicate with other tasks. This model is transparent to the application and it even allows the combination of heterogeneous resources to be seen as one virtual machine from the application's point of view.

[†] The work presented in this chapter was done while both authors were at Paderborn Center for Parallel Computing, http://www.upb.de/pc2

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 239-248, 1999
© Springer-Verlag Berlin Heidelberg 1999

The reasons for the popularity of PVM can be found in its specific features: PVM supports a dynamic process model, it provides mechanisms for fault tolerance, it has a manageable number of message-passing functions, it is available for a wide variety of computer architectures, and it even runs on heterogeneous clusters.

Virtual machine and application control is brought to the user level by means of daemons (`pvmds`) running on the hosts of the distributed system. The daemons are responsible for process control and for message routing.

Another important part is the PVM library (`libpvm`) which performs the actual message passing on the underlying hardware. For each PVM execution environment there exists a `libpvm`. The sending and receiving tasks are identified by PVM task identifiers (`tids`).

The standard PVM package supports two routing policies: With the *default routing*, a message is sent to the local daemon which forwards it to the destination daemon that serves its local task. The data transmission to the local daemons is done with TCP, whereas the communication between the two remote daemons is done via a connectionless UDP protocol. This approach gives the necessary flexibility allow communication with a wide variety of different systems.

The faster *direct routing* scheme establishes a direct connection between the two user tasks without involving the daemons. This is done with the `pvm_setopt(PvmRoute,PvmRouteDirect)` directive that invokes the PVM library to set up a direct TCP connection between the two user tasks.

In our PVM-SCI implementation, we have modified the PVM library to send messages with `PvmRouteDirect` via the fast shared memory communication provided by SCI, as presented in Section 13.4.

13.2.1 PVM Implementations

PVM owes much of its popularity to the fact that it runs on a wide spectrum of target platforms. This includes all major operating systems and a variety of communication protocols. Its special feature is then to be able to interoperate between these heterogeneous environments.

Most current LAN or WAN implementations of PVM use the IP stack and UDP or TCP as their fundamental protocol layer. One such example is a PVM 3.3 implementation for ATM networks [11] that uses TCP even when the `PvmRouteDirect` option is set. On the one hand the use of the full software protocol stack has the advantage that PVM can be quickly adapted to support new interconnects, but on the other hand the short software development time and improved portability is often payed for by a much reduced communication performance.

For the faster system area networks (SANs) *Myrinet* [3] and *SCI* [6], some recent message-passing implementations try to by-pass the costly system calls and access the network directly. Research projects on *Active Messages* and *U-Net* [1] showed improved communication performance. The ongoing trend

since then can be seen to enable user-level communication without involving the operating system kernel. Following this line, the *Virtual Interface (VI) Architecture* [10] has been proposed as a standard. Some hardware vendors already support VI Architecture, and there also exists a first MPI implementation [7].

Other related work has been done by developing a PVM-compliant version running on the *ParaStation* [2]. Here, the original PVM source code has been modified thoroughly to achieve maximum communication performance on the underlying Myrinet. But compared to PVM-SCI, the *ParaStation User-Level Communication (PULC)* runs on homogeneous environments only.

With our work on PVM-SCI we aimed at both, to provide maximum communication performance over SCI, and to support transparent downgrading to other communication media when SCI is not available. In SCI clusters, the computing nodes are typically interconnected by Ethernet for resource management purposes, while the faster SCI links are used for application level communication only. We devised PVM-SCI to make use of SCI for communicating within the cluster and to use Ethernet or other suitable communication media for connecting to nodes outside the SCI cluster.

13.2.2 Models for Zero-Memory-Copy Data Transfer

Traditional networking schemes (for example TCP as a form of network transport layer) move data as streams: the user task moves data from its local memory into a buffer and calls a library routine to hand the buffer over to an I/O interface for transporting the data as a byte stream to the communication network. At the receiving side, the bytes arrive and are filled into a buffer, the buffer is handed over to the operating system, which, awakened by an interrupt, provides the data to the user task. After a context switch, the receiving user task copies the data into its local memory for further processing.

While this state of affairs might be tolerable for transferring data over WANs or LANs, the described scenario is unsatisfactory in SANs. Due to their faster communication speed, the time used for a context switch or for a memory transfer is no longer negligible compared to the actual data transfer time. Here, not only the external communication network may become a bottleneck, but also the data transfer through the memory bus. On a 200 MHz Intel PentiumPro, for example, a memory read is done at a speed of about 180 MByte/s, while a `memcpy` is only half as fast.

Consequently, schemes for zero-memory-copy data transfers have been devised [8, 9], however at the cost of sacrificing some compliance with message-passing standards. *BIP-MPI* [8], developed at ENS Lyon, is an adaptation of MPICH for zero-memory-copy on Myrinet. With 126 MByte/s on large data blocks, its communication performance almost matches the 133 MByte/s throughput of the 33 MHz PCI bus.

13.3 SCI Communication Model

SCI supports a shared memory communication model by making a process' local memory available to other processes via the SCI interface card. From an application's point of view, the communication layer is responsible for exporting local memory pages to other participating processes. The driver software for the SCI adapter card takes care of data retrieval and delivery. Compute nodes that wish to participate in the sharing of the common address space must map the memory chunks, thereby implicitly building up a connection.

In SCI such a connection is a mapped memory segment (also called a *window*), which has been set up by allocating and exporting memory by one process and importing it by another. Depending on the mapping type, a segment can be mapped in either read-only or write-only mode. Since remote writes are much faster than remote reads (approx. 66 MByte/s versus 8 MByte/s), we have implemented PVM-SCI in such a way that data is only read from local memory segments which are exported with write permissions to remote processes.

Communication Setup. In the setup phase of an SCI connection, three basic parameters are used by the Scali SCI driver (9) for allocating, offering, and mapping memory to enroll into the SCI address space. The first parameter is a unique `HostID`, which is obtained by calling `SciGetNodeId()`. The second parameter is a `ModuleID` for each task, which should be nonambiguous during runtime. Finally, a `ChunkID` needs to be provided which is unique within a task assignment. This way, different tasks on one node can allocate segments without interfering with other allocations.

Having determined these parameters, the memory allocated by `SciAllocateLocalChunk()` can be offered for remote access by passing the parameters to `SciOffer()`. The protection method of the virtual mapping must be defined either for exclusive read or write, or for a combination of both. Other nodes can then connect to the memory by calling `SciConnectToRemoteChunk()` with the same parameters.

Thus an exchange of basic information is needed in advance. Since no control flow is given at the lowest SCI level, the initial information is exchanged via Fast Ethernet, which is available in most clusters as a basic communication medium for cluster setup and management.

After having established a connection, the processes are able to transfer data. The arrival of a new message can be signaled to the communication partner by an interrupt mechanism consisting of an array of flags which can be set by remote processes.

The Scali SCI API provides two mechanisms for checking the interrupt flags, a polling method and a sleeping variant. Alternatively, one can add additional message control at the application level (without using interrupts) by adding a counter of the messages sent and received.

13.4 PVM-SCI

13.4.1 System Architecture

The standard PVM implementation uses connectionless UDP/IP sockets as the fundamental inter-processor communication scheme. This requires three communication hops (Figure 13.1): first, a UNIX socket communication between the sending task and the local `pvmd`, then a connectionless UDP/IP communication between the two `pvmd`s, and finally again a UNIX socket communication to the receiving task. This mechanism works on any combination of system environments. In homogeneous environments, a direct TCP/IP route can be set up with the `PvmRouteDirect` option, which bypasses the two daemons.

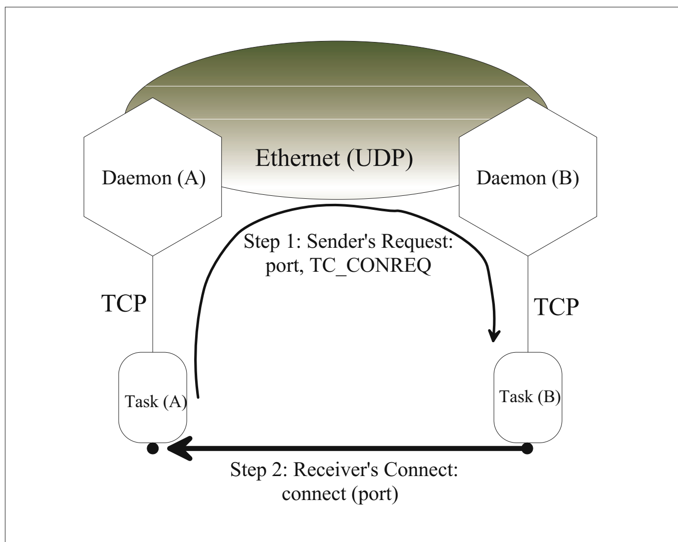


Fig. 13.1. Protocol for establishing a task-to-task connection with standard PVM

In PVM-SCI the direct routing is further improved by sending data via the SCI shared memory facilities. For this purpose, PVM-SCI needs to know how to address the SCI memory that has been exported by the communication partner. This information is negotiated between the controlling `pvmd`s at both sides as illustrated in Figure 13.2. In contrast to standard PVM where a socket provides a bidirectional connection, two separate windows (each of them allocated by one task) must be created to allow the sending and receiving of messages with the fast remote write and local read functions.

Let's assume that task A wants to send a message to task B for the first time with the `PvmRouteDirect` option set. Task A then allocates, maps, and offers a local memory chunk identified by a unique `NodeId`, a `ChunkId`, and

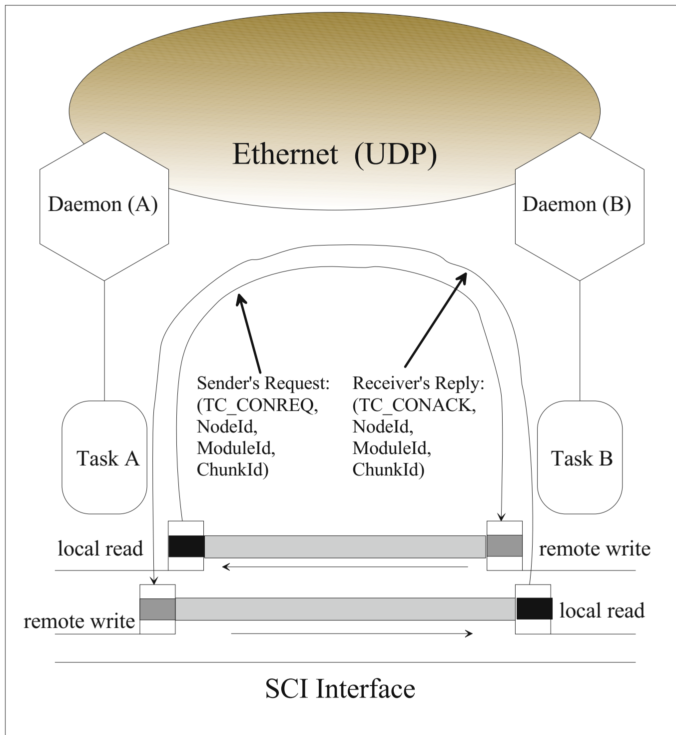


Fig. 13.2. PVM-SCI protocol for establishing a task-to-task connection via SCI

a `ModuleId`. This information is destined for the remote task B. It is sent to the remote `pvmd`, which notifies the destination task B when a `receive` or `send` function is being invoked by task B. The three parameters (`NodeId, ChunkId, ModuleId`) are used to authenticate a connection with the correct memory chunk. Authentication is necessary because multiple windows may be offered by the same task.

Likewise, task B allocates, maps, and offers its local memory chunk to task A so that, at the end, a symmetrical connection is built up with two memory chunks at both sides. This allows both tasks to read locally and write remotely when sending data to each other.

In general, a communication request is granted when (1) the routing policy and implementation allow for a direct SCI connection, (2) the resources are available, and (3) the protocol versions match each other. If not, a `request-denied` reply is returned and the default TCP connection is used instead.

When the connection has been established, the actual message transfer is done with the `pvm_send()` command. Task A copies the packed message buffer into the current write position of its mapped write window, updates

the number of messages sent to task B, updates its current write position and triggers the notification flag.

When entering a receiving function, task B detects a message arrival by querying the interrupter. If no message has arrived yet, it will sleep on the interrupter until the flag is set. It then reads the header of the incoming message from its current reading position in its read window to get the size of the message. When the message has been taken out of the window, the current reading position is updated. This variable resides in the mapped memory and is therefore concurrently updated on the remote side. This is necessary to inhibit future write operations from overwriting unread messages in the ring buffer.

Note that PVM-SCI does not establish the connections at program startup time, but only after the first send with `PvmRouteDirect`. Thus, the memory is allocated and mapped just before being used, thereby reducing the use of resources and speeding up the program startup time on large systems.

13.4.2 Supporting Multiple Interconnects

One advantage of the standard PVM model is often seen in its support of heterogeneous computing environments. PVM-SCI even supports heterogeneity at two levels: the computing nodes and the interconnects. With PVM-SCI it is possible to run applications in heterogeneous computing environments with any combination of SANs, LANs, or WANs. The fast SCI interconnect is used locally while PVM-SCI downgrades to conventional Ethernet or ATM networks when communicating to the outside world.

The standard PVM uses the `select()` function to wait for incoming messages on a set of available connections (typically the TCP connected daemon and communicating partners). With additional interconnects, PVM-SCI has to pay attention to multiple interfaces. While a threaded implementation with multiple threads serving multiple interfaces would have been probably the most elegant method to implement, we observed poor performance caused by the additional synchronization steps and context switches. We therefore implemented an active polling method which checks for messages on each available interface. To avoid busy waiting, the UNIX system call `sched_yield()` incrementally increases the relinquishing value before re-entering the loop.

13.4.3 Reducing Memory Copies

In some systems, the bandwidth of the local memory bus may become the limiting factor when transferring data from memory over a SAN to a destination processor. To make things worse, the standard PVM software distribution reads the message buffer twice: the first time to obtain the header of a message with basic information like tags, source, destination, and message length, and the second time to retrieve the data.

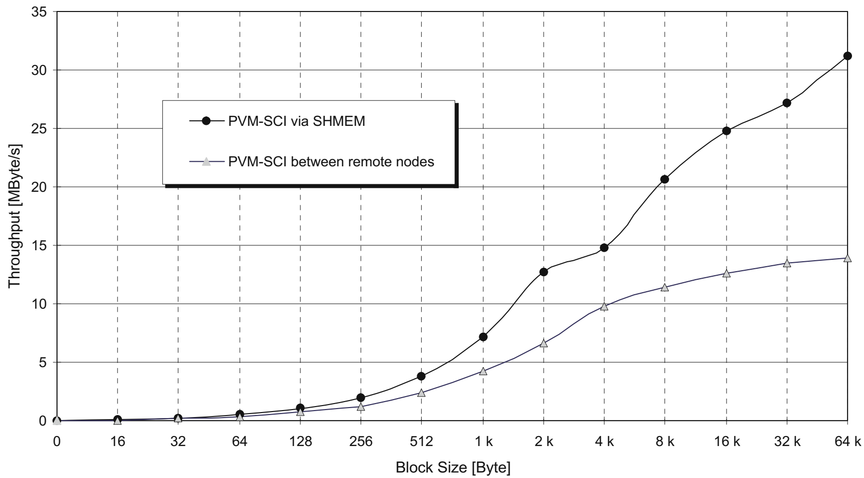


Fig. 13.3. PVM-SCI performance on Linux (100 kByte window size)

PVM-SCI tries to retrieve the message in one step using a *one-receive-fits-all* strategy. Instead of reading the header first, then determining the size of the message body and thereafter retrieving the actual data, our method tries to read the message in one step. In more detail, the number of bytes read are the header size plus a constant number of bytes reflecting the assumed maximum number of bytes in the message which are read out of the mapped memory and copied into a message buffer. This value is currently set to 200 bytes. When the message body contains less than 200 bytes some bytes at the end of the allocated buffer have been read in vain. In the other case, i.e. when there are more than 200 bytes, a second `memcpy` completes the receive function. The estimated message length may be set by the application. Currently we investigate a method for offering an additional `pvmsetopt()` option which can be set by the application to a typical message size.

13.4.4 Ring Buffer Management

The SCI API provides basic mechanisms for setting up a connection between the processes, but it does not support the message handling itself. Specifically, there are no functions for changing the size of previously allocated memory chunks. We have therefore implemented a ring buffer protocol with different windows for every send/receive combination. The actual ring buffer implementation is very similar to the one used in the SSLib project, therefore we refer the reader to Chapter 11 for a detailed discussion of this topic.

13.4.5 Performance Results

Figure 13.3 shows the throughput of PVM-SCI for messages sent with the `PvmRouteDirect` option. The performance has been measured with Linux on 400 MHz Pentium II PCs with Intel BX chip-sets. The upper and lower graphs in Figure 13.3 show the throughput for intra-node and inter-node communication, respectively.

With local communication, PVM-SCI reaches a peak performance of 32 MByte/s at a block size of 64 kByte. Increasing the block size does not yield any further improvement. With remote communication, the maximum performance drops dramatically down to 14 MByte/s. The round trip communication latency is 85 μsec. These performance figures are much worse than those of ScaMPI (Chapter 14), which is attributed to the time-consuming interrupts (not used by ScaMPI) and to the overhead of the `memcpys`. We currently investigate methods to avoid the use of interrupts and we also plan to utilize the fast Pentium MMX routines for memory copies.

Note that half of the peak performance is achieved with relatively small messages of about 3 kByte. This is a typical feature of SCI, which supports the fast transmission of small packets.

13.5 Conclusions

PVM-SCI is a complete implementation of the PVM standard for SCI. With the `PvmRouteDirect` directive set, PVM-SCI utilizes the fast shared memory communication primitives of SCI. When SCI is not available, PVM-SCI automatically downgrades to the default TCP protocol without the need to change the application code.

PVM-SCI is fully compliant with the latest and probably final PVM release 3.4. The architecture has been kept open and modular to allow for easy adaptation to other interconnects, such as Myrinet or ATM for example. PVM-SCI initially tests the availability of interconnect adapters, starting with the fastest (SCI), then trying ATM and finally checking for Ethernet.

This adaptive degradation is also used in the follow-up message-passing environment Harness (Heterogeneous Adaptable Reconfigurable NEtworked SystemS) [5]. The Harness project aims at combining and integrating network layers such as Myrinet and SCI, as well as conventional network interconnects (e.g. ATM). Other communication paradigms, like Active Messages or shared memory may also co-exist on the cluster, and can be plugged in on demand. PVM-SCI has been designed modularly so that it can be plugged into Harness as soon as it becomes available.

In its current implementation, PVM-SCI is still too slow. We therefore examine methods to avoid the costly interrupt mechanisms (as done in ScaMPI) and to use the fast MMX functions on the Intel processors for improving the memory copying.

References

1. A. Basu, M. Welsh, and T. v. Eicken. Incorporating Memory Management into User-Level Network Interfaces. *Proc. Hot Interconnects V*, Stanford University, CA, August 1997.
2. J. Blum, T. Warschko, and W. Tichy. PULC: ParaStation User-Level Communication. Design and Overview. *Proc. IPPS/SPDP 98*, Orlando, FL, March 98.
3. N. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro* 15(1), pages 29–36, Feb. 1995.
4. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, Boston, 1994.
5. *Harness: Heterogeneous Adaptable Reconfigurable NEtworked SystemS.* http://www.epm.ornl.gov/harness.
6. IEEE Std 1596-1992. *IEEE Standard for Scalable Coherent Interface (SCI).* Inst. of Electrical and Electronics Eng., Inc., New York, NY, August 1993.
7. MPI Software Technology Inc. Homepage. http://www.mpi-softtech.com.
8. L. Prylli and B. Tourancheaux. BIP: a New Protocol Designed for High-Performance Networking on Myrinet. *Proc. IPPS/SPDP 98*, Orlando, FL, March 98.
9. H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-Down Cache: A Virtual Memory Management Technique for Zero-Copy Communication. *Proc. IPPS/SPDP 98*, Orlando, FL, March 98.
10. Virtual Interface Architecture. Homepage. http://www.viarch.org.
11. H. Zhou and A. Geist. *Faster Message Passing in PVM.* Technical Report, Oak Ridge National Laboratory, Oak Ridge, TN, 1995.

14. ScaMPI – Design and Implementation

L.P. Huse, K. Omang, H. Bugge, H. Ry, A.T. Haugsdal, E. Rustad

Scali AS, Norway
Email: {lph,knuto,hob,hwr,ath,eir}@scali.no

14.1 Introduction

MPI (Message Passing Interface) [7] is an established and de-facto standard for information exchange based on the message-passing paradigm. MPI was standardized by the MPI Forum in 1996. Academia, industry, and vendors of high-performance computers drove the effort.

The MPI has a rapidly growing community as a standard user API (application programming interface) for parallel programming. Today, MPI is the preferred API for portable, parallel programs, and the success of the standard can be illustrated by applications running on both shared and distributed memory systems. Examples of the former are systems from SGI and Sun, whereas IBM, Cray (now SGI), and Scali deliver systems adhering to the latter category. Applications written using MPI are deemed very portable, and they can easily be ported between shared and distributed memory systems. Thus, seen from an ISV (independent software vendor), a message passing application is more portable than an application using the shared memory paradigm, since the message passing application can run on either shared or distributed memory systems.

In the rest of the chapter the design and implementation of ScaMPI, Scali's high performance MPI implementation is presented. Key technical achievements of ScaMPI are low latency, high bandwidth and flexibility of transport medium as well as options for speeding up application performance within SMPs by allowing the use of threads. The programming environment for ScaMPI provides various built-in options for debugging and tuning. In addition ScaMPI is integrated with powerful third party software. Performance of important ScaMPI primitives is discussed in light of recent performance measurements. These measurements also document excellent scalability of ScaMPI for up to 96 inexpensive dual CPU nodes.

14.2 Scali Systems

Scali systems use SCI as interconnect. To make affordable systems, Scali has chosen to use standard I/O (input/output) buses, such as PCI [13] and Sbus [5], as attachment point to the SCI interconnect fabric. SCI has specified cache coherency as an option, but since the I/O buses for most workstations

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 249-261, 1999
© Springer-Verlag Berlin Heidelberg 1999

are decoupled from the main memory bus by an I/O bridge (see Figure 14.1), the I/O bus cannot intercept a processor accessing the local memory.

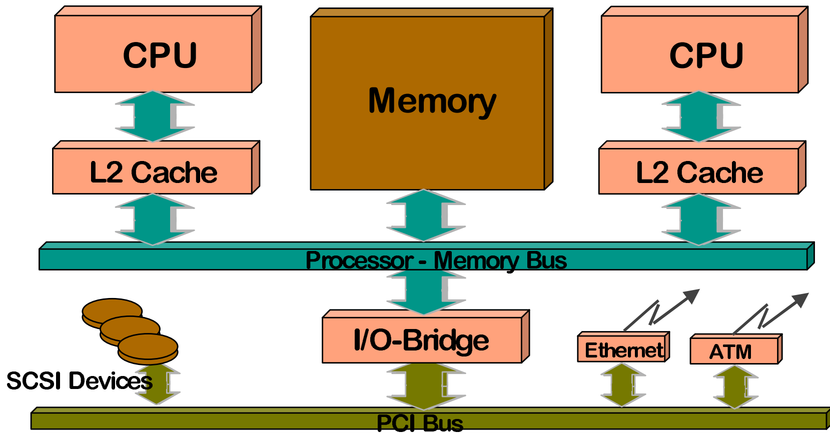


Fig. 14.1. Block diagram of a node

This inhibits a global cache coherent memory model to be implemented in hardware. A clear benefit of using the I/O bus as attachment point is that the I/O buses are standardized. For the IHVs (independent hardware vendors) this means having a larger market for their products, and larger volumes again imply lower prices. Also, the evolution of new generations of I/O buses, e.g., faster clock frequencies, wider buses etc., is less frequent compared to the evolution of processors and their accompanying buses. As a consequence, the cost for peripheral boards tends to be low compared to special, proprietary hardware components.

14.3 The SCI Memory Model

The cache coherency in SCI is documented in detail. The memory consistency model on the other hand, is left to the implementors. Thus, any memory consistency model, such as the sequential, processor, weak, or release consistency model might be implemented. This applies to systems using either the I/O or the cache coherent processor bus as the attachment point. Software running on SCI based systems must therefore either be explicitly or implicitly aware of the memory consistency model provided by the system.

Systems using the I/O bus as the attachment point are in most cases distributed memory systems, where each node runs its own instance of the operating system. Such systems, e.g., SCI based, are able to provide a shared address space programming model, where portions of the virtual address space of one process can be made visible to a process running on another

node. The conceptual simplicity of this model compared to the traditionally layered ISO/OSI model can easily be illustrated. For two user level processes running on different nodes to communicate, both need to map the same SCI shared memory segment into their virtual address spaces. The SCI shared memory is for performance reasons always physically located on the receiver side. For two processes to communicate, the sender process writes data to the remote memory segment and the receiver reads the data from local memory, without any system calls or expensive protocol processing.

Middleware for this shared address space model must be explicitly aware of the underlying memory consistency model, which is influenced by the characteristics of the processor, the I/O bridge, the SCI adapter, and the SCI interconnect fabric. For example, common hardware techniques for performance enhancements, such as write buffering, write combining, and prefetching might be implemented in multiple of these hardware components. To Scali, this illustrates two problems with the shared address space model: the user must have an intimate knowledge of the hardware components, and very few applications are written for the shared address space model.

14.3.1 Coordinating Use of Shared Locations

There are several ways of using shared locations to exchange data as needed to implement message passing. One well known technique is to use the concept of *critical regions* where only one process at a time has access to a particular shared resource. An example from MPI is how to implement buffer allocation. With mutual exclusion, an implementation can allow multiple sender nodes to allocate receive buffers at a particular remote node from the same buffer pool. This technique has been frequently employed in MPI implementations for SMPs. The approach is simple and makes it easy to implement space efficient resource management policies and can be implemented with efficiency between processors on the same memory bus. However, as the number of communicating processes grows, contention for the involved locks may hurt performance.

Efficient implementation of mutual exclusion locks requires atomic memory operations. Atomic operations such as compare-and-swap and various forms of fetch-and-op are available (or can be built with the available basic atomic primitives) in most modern system bus architectures. The SCI standard specifies a number of such operations, but only a very limited atomic operation support (fetch-and-increment-by-one) is available for the current Dolphin PCI/SCI implementation [2].

Efficiency is also complicated by the fact that remote loads (fetching data over the network) are an order of magnitude slower than local loads. To achieve the atomicity provided by the PCI/SCI hardware in absence of properly implemented locking primitives on the PCI bus, all accesses to the lock memory must go through the PCI/SCI hardware, i.e., they are remote accesses from a performance perspective [10].

A simple solution where locks are avoided is to make access to shared locations disjoint with respect to stores, that is, only a single process has the right to store to a particular location. For the example on dynamic buffer allocation in the receiver, this means that there must be separate buffer pools for each sender at each receiver. Thus this may look like a speed-at-cost-of-space trade-off. However, with the option of allocating the buffer pool for a particular sender at a particular receiver only when needed, the extra buffer capacity spent can be kept low.

14.3.2 Ensuring Safe Data Transport in SCI – Checkpointing

Usually when transferring data from one node to another over a SCI network, the data arrives at its destination fast and accurate. However, in a multi-node shared memory environment there is always the possibility of nodes being temporarily unavailable (e.g., due to high priority OS calls), data alteration in the network due to electronic noise (detected by CRC checks), the I/O bus may be occupied with other high priority traffic etc. All of these events are detected in Scali systems, and those that are related to the SCI network are corrected by the SCI driver. To make certain that the data arrives correctly over the network, checkpointing needs to be employed.

Checkpointing is a common programming technique used in systems where dynamic errors may occur, e.g. in shared memory and database systems. Before and after all operations sufficient status information is gathered to check if the operation was completed successfully. If the checkpoint fails the effects of the faulty operation have to be nullified and the operation has to be repeated. For SCI, the checkpoint procedure is initiated by first flushing all data onto the network. The checkpoint state is then derived from the state of the driver and an interrupt counter. If the checkpoint state changes during a data transfer, the data has to be retransmitted. A shadow of all necessary driver information is mapped into user space for fast checkpointing. An example of using checkpointing in shared memory programming is given in Figure 14.2.

14.3.3 Shared Address Space Programming without the Drawbacks

To enable users to exploit the benefit of the shared address space model without detailed hardware knowledge, standardized APIs are provided to the application programmers. A large existing base of applications can then be used directly (without modification of even old "dusty deck" applications), just by recompiling. The applications are thus able to communicate efficiently, without being burdened by excess copying, system calls, interrupts etc. as would have been necessary if a typical software stack of the traditional ISO/OSI model had been used. Today, MPI [7], PVM [4], Fast Messages [11],

and Split-C [1] are also available on Scali systems. ScaMPI is Scali's own high performance implementation of MPI, the others are available through various academic institutions.

14.4 ScaMPI Design Goals

The following goals were set forth in the design process of ScaMPI:

Scalability: System size ranging from one to hundreds of nodes should be supported.

Low latency: We aimed at message latency around 10 μs for exchanging MPI messages from user level to user level. The latency of collective MPI operations should grow with $O(log(N))$, where N is the size of the system.

High bandwidth: Point-to-point bandwidth should be close to the theoretical maximum for the actual implementations. The bandwidth available to each node performing MPI collective operations should be constant and not be reduced with increased system size.

Fault tolerance: The SCI interconnect fabric might be subject to errors, such as CRC errors, cables being unplugged etc. Such transient errors should be transparent to applications. For example, it should be possible to exchange cables while an application is running, without affecting it except for reduced performance. Another example is that it should be possible to change the routing function in the SCI interconnect fabric, i.e., the communication paths, transparent to the running application.

Flexibility of transport medium: Although SCI shared address space was intended as the primary transport medium, we aimed at leveraging this implementation and support true shared memory as the transport medium for MPI processes communicating on the same SMP node. The selection of the actual transport medium should be automatic and transparent to the user.

User friendliness: To ease application development we aimed at providing different levels of startup procedures to accommodate different requirements, such as debugging, profiling, logging etc.

Thread-safe implementation: ScaMPI must support different mappings of MPI processes to hardware resources. In a one-to-one mapping each MPI process is mapped to its own CPU, while in the one-to-many model each MPI process is mapped to a set of CPUs - typically all the CPUs in a node. In the one-to-many model, explicit multi-threaded programming or an automatic parallelization tool is used to efficiently exploit all system resources. Here different threads constituting a single MPI process might simultaneously request services from the MPI library. Thus, ScaMPI as well as the SCI middleware had to be designed thread-safe in a way that enables a high level of parallelism.

14.5 ScaMPI Implementation

ScaMPI was designed to take advantage of SCI's shared address space architecture. The focus has been on utilizing those features ensuring the best possible performance, both with respect to latency and bandwidth. A write-only protocol [3] was chosen for two reasons. First, performance of remote writes are better than remote reads, as described in [10]. Furthermore, using a write-only protocol ensures cache coherency, even though the attachment point is the I/O bus, as discussed in Section 14.3. Since reading data from local memory is much faster than fetching it over the SCI network, ScaMPI always use a remote-write-local-read policy. This contributes significantly to fulfill one of the design goals of ScaMPI: to be able to scale performance to very large systems.

14.5.1 Fault Tolerance

Three important items are required to securely manipulate data structures on a remote node:

- Atomicity of multi-byte entities must be controlled. This implies that either all or nothing of a multi-byte entity is modified, i.e., that it is never partially modified. Consider for example a simple ring-buffer structure with a write and a read index. The receiver polls the sender's write index, and compares it to the read index. If the two indices differ, the ring buffer contains valid data. If the write index is represented by a two-byte entity, and if those where updated one at a time, catastrophic errors could be the consequence when the index wraps from the least to the most significant byte.
- Enforcing memory consistency, i.e., ensuring that all previously issued write requests have been globally performed. For example, if one process writes a block of data to a remote buffer, and then signals the completion of the transfer by writing to a flag in the receiver node memory. If the memory consistency is not enforced between the data transfer phase and the signaling phase, the receiver might consume stale data.
- Error checkpointing. As discussed in Section 14.3.2, a transfer might have been corrupted, e.g., the cable has been unplugged. A methodology of checkpointing is needed to ensure correct data transfers.

The natural sequence of operations to securely transfer a data block and set a flag at the receiver is depicted in Figure 14.2.

Scali's SCI driver, ScaSCI, has combined the functionality of enforcing memory consistency and checkpointing, to improve speed. As illustrated in Figure 14.2, memory consistency has to be enforced before the checkpointing routine is called. Otherwise, active outstanding write operations might be in progress when the `EndCheckPoint()` routine is called, and those might later be exposed to errors. Another important observation from Figure 14.2 is that

```
void SendMsg(long *remoteFlag, void *remoteDst,
             void *localSrc,   int   sizeOfMsg)
{
   CheckPointToken token;
   StartCheckPoint (&token);
   do {
        Memcpy(remoteDst, localSrc, sizeOfMsg);
        MemBarrier();
   } while (EndCheckPoint(&token) != SUCCESS);

   StartCheckPoint (&token);
   do {
        *remoteFlag = SUCCESS;
        MemBarrier();
   } while (EndCheckPoint(&token) != SUCCESS);
}

void RecvMsg(long *localFlag, void *localSCIMem,
             void *localUser, int   sizeOfMsg)
{
   while (*localFlag != SUCCESS) {
      sleep();
   }
   Memcpy(localUser, localSCIMem, sizeOfMsg);
}
```

Fig. 14.2. Pseudo-code for secure one-way data transfer

the side effect exposed to a remote memory region might be exposed more than once, in case of error indications. The SCI responses might have been subject to errors, and not the requests. If this is the case, the side effect has taken place, but the requester node cannot distinguish between a failure of a request and of its response. Therefore, the requester node has to re-issue the data transfer. As a consequence, the data structures used in ScaMPI had to be designed idempotent. A data structure being idempotent will be consistent even if an update was carried out more than once.

Another important aspect from Figure 14.2 is that messages actually are securely transferred to the remote node, before the sender is able to signal to the receiver that the messages are ready for consumption. The time spent to enforce memory consistency and to perform the checkpointing will be directly added to the latency of transferring a message. The impact will be more severe the smaller the message is. To avoid this added latency, ScaMPI has a combined message data structure for small payloads, the *MPI message envelope* [7, Section 3.2.3] and a field, *ready*, indicating to the receiver that this structure represents a new, unconsumed MPI message. ScaMPI uses 64 bytes to represent this information, including 32 bytes of MPI data payload. As discussed above, atomicity, or merely lack thereof, must be handled. Since few processor instruction sets have provisions for 64-byte atomic store operations, a mechanism to prevent the consumer from receiving a partly received message had to be found. Even if the sender specifically wrote the *ready* field as the last part of the transfer, the data could appear at the receiving node in a different order, due to the possibility of reordering of packets in the SCI

interconnect fabric. To avoid this pitfall, ScaMPI has included a CRC check value in the structure to protect its integrity. This approach enables ScaMPI to send self-synchronizing messages in a safe way. As an additional bonus, the receiver may read the message while the sender completes enforcing memory consistency and performs the checkpointing, thus reducing the latency.

14.5.2 User Friendliness

Scali has put an effort into making ScaMPI and its environment user friendly. The execution of an MPI-program is started and controlled by a monitor program (`mpimon`). The monitor takes two types of parameters on the command line:

Parameters controlling the ScaMPI set-up. These parameters include customizing the set-up of SCI memory allocation, buffer sizes, barrier fanin/fanout etc. The parameters are checked for validity and, if not correct, the program execution is aborted and appropriate error messages are given. Being tuned for performance, ScaMPI by default allocates buffers the first time a communication channel is used. For performance measurements and communication debugging, ScaMPI can be set to initialize all the communication channels at startup time.

MPI program names, their parameters and node specifiers. The parameters are automatically distributed to all processes constituting the parallel program, not only process zero. The node specifiers are checked for legal node names. Each node can occur several times within a node specifier, enabling more than one process per node. For full control over process-to-node mapping, ranks are allocated sequentially from the node specifiers.

ScaMPI has the ability to have multiple MPI programs in one run. This is specified by adding multiple blocks of MPI programs, parameters and node specifiers on the command line to the monitor.

Input from the user (`stdin`) can be distributed to all or some of the processes. Output from the processes (`stdout` and `stderr`) is displayed in the window where the monitor was started. All processes inherit the running environment from the monitor shell, e.g., the current directory path, i.e., where the monitor was started. The user can choose to have none, some or all environment variables copied to the processes.

Scali has added some functionality to ScaMPI and the monitor to ease debugging and to get a better overview of what is happening when running an MPI program. Output from selected nodes can be printed in separate windows or files. MPI programs can selectively be started within a separate window or debugger to allow use of other debug/trace tools.

14.5.3 Third Party Software

From a user perspective, a Scali system interface is a parallel tools environment. The parallel user environment consists of three basic components,

system access control, parallel debugger and parallel performance analysis. The purpose of a parallel tools environment is to create a single system image of a parallel computer. It is however not possible to shield the user completely from the added parallel complexity needed to get more computational power. But a high quality parallel programming environment contributes significantly to reduce the time spent in developing and debugging code.

Any standard MPI parallel tools [15, 8] can be used with ScaMPI. Since a Scali system is built from COTS (commercial off-the-shelf) technology by using standard hardware and software components, third party parallel tools by any independent cluster vendor can be used on a Scali system running ScaMPI. To provide the ScaMPI user with basic state-of-the-art parallel tools, ScaMPI is available with the TotalView [16] parallel debugger and the Vampir [12, 14] parallel performance analysis tools.

The TotalView [16] graphical parallel debugger has support for the most important parallel programming models: threads, MPI, PVM, and HPF. Supported platforms come from the major supercomputing vendors including Compaq, Digital, SGI, IBM, and Sun. The main parallel feature is the single point of control for debugging ScaMPI programs. From a single window it is possible to control individual groups of processes, hide unnecessary and display essential information. On startup, TotalView gives the user an option to stop in `MPI_Init()`, the starting point of any MPI program. After this initial stop, the user can set appropriate action points before continuing the parallel debug session. TotalView has a fast and intuitive GUI, with the possibility of data visualization, a useful aid in debugging numerical programs. TotalView is designed for multiprocessing and offers the debugger features a programmer expects.

Vampir [12] is a tool for performance analysis of MPI programs. In the NHSE (National HPC Community Software Exchange) parallel tools review [9] Vampir was rated as the best parallel performance tool. It is available on all major supercomputer platforms. To collect performance data, the ScaMPI program is linked with the VampirTrace library, and run. The performance data is logged to a file for post-processing. Vampir ScaMPI performance analysis helps the user organize the performance data, understand application and communication behavior, evaluate load balancing, and identify communication hot-spots. A very useful feature is the extensive space-time filtering of data to extract relevant information only. A time-line window displays application and message passing activities and shows parallelism as the sum of active non-communicating processes. Communication statistics can be displayed for selected intervals of time and message length.

14.6 Performance Results

The tests were run on a 96-node system with dual Pentium II 450 MHz processor PCs interconnected with PCI-SCI cards from Dolphin ICS [2]. The

SCI network is organized as an 8 x 12 2D mesh/torus. ScaMPI delivered a 9.4 μs ping latency and up to 76 MByte/s between two MPI processes on separate nodes over the SCI network (4.5 μs and 130 MByte/s between two processes on the same PC). This fulfills two of the ScaMPI design goals (Section 14.4), low latency and high bandwidth.

As stated in the design goals for ScaMPI, latency of collective MPI operations should grow with $O(log(N))$, where N is the size of the system, while bandwidth per node should be nearly constant for all system sizes. Restrictions of the bandwidth in multi-dimensional tori is analytically calculated in Chapter 8 and indicates the feasibility of this goal. In the next two subsections the achievement of these goals will be shown through practical measurements. To get comparative results the test programs were run on a far more expensive state-of-the-art 128-processor (MIPS R10k) Cray Origin 2000 equipped with 192 MByte main memory.

14.6.1 Barrier

Barrier is a collective operation that carries no data, but synchronizes all processes. Since barrier does not carry any data, it is a good measure for the collective latency. ScaMPI's barrier implementation uses a fixed fanin/fanout tree and operates directly on SCI shared memory.

Nodes	2	4	8	16	32	48	64	80	96
Timing	8.1	8.2	9.3	20.6	24.4	26.1	29.9	30.8	33.1

Table 14.1. Barrier performance in μs

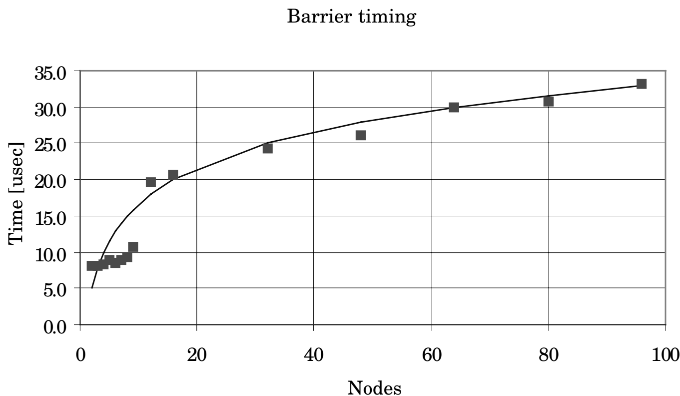


Fig. 14.3. Barrier performance compared to $const * log(N)$

Table 14.1 shows absolute timing of barrier over the SCI network. As can be seen, this is a very fast implementation with sub-latency performance up to 8 nodes! A barrier involving two processes on the same PC use only 1.4 μs. The Origin 2000 used 25.9 μs to synchronize two processes (739 μs for 64 processes) [6]. Figure 14.3 shows graphically how the performance results of the barrier compare to a $const*log(N)$ trend. As can be seen, ScaMPI over SCI shows a good match of the timing of a barrier, i.e., collective latency, and a $const * log(N)$ trend.

14.6.2 All-to-All Communication

The most demanding communication situation for a machine is when all nodes communicate with all the other nodes. Performance of `MPI_Alltoall()` is therefore a good measure of the aggregate bandwidth of a system.

Nodes	2	4	8	16	32	64	96
Throughput	30.9	35.2	36.5	31.1	32.3	31.7	26.2

Table 14.2. All-to-all communication performance per node in MByte/s.

Table 14.2 shows measured communication performance per node of all-to-all communication for long messages over the SCI network. For two processes on the same PC throughput was measured as 54.6 MByte/s. The SCI based system shows far better scaling than the Origin 2000, which delivers a per-node performance of 42.1 MByte/s between 2 processes, 26.9 MByte/s between 16 processes and ends up with a mere 7.3 MByte/s for 32 processes [6]. The very poor performance scaling on the Origin 2000 may have to do with interference from other user programs due to lack of resource reservation.

The all-to-all performance is calculated on the basis of the network traffic. Since `MPI_Alltoall()` uses two buffers, an N-th part of the send buffer is copied internally to the receive buffer and is therefore not part of the network data volume. For all-to-all communication between two nodes, half of the data is transferred and the other half is copied. This is the reason for the apparently low performance between two nodes. The shift in performance between 8 and 16 nodes is caused by a change of the algorithm. If the algorithm for larger configurations had been used for all configurations no performance shift would have appeared, but the performance for small configurations would have suffered. For up to 8 nodes, the SCI network delivers sufficient throughput for all nodes to communicate at full speed. This is compliant with the constant-per-node performance of up to $64 (= 8*8)$ nodes and a small decrease for 96 nodes. By going from a 2D to a 3D torus network, the interconnect performance should scale "perfectly" to $8*8*8 = 512$ nodes. This conforms to the conclusions in Chapter 8.

14.7 Conclusions

Our initial ambitions were to make a thread-safe, scalable, low latency, high bandwidth, fault-tolerant, user friendly and flexible (with respect of transport medium) implementation of the MPI standard. ScaMPI is meeting all of these design goals.

The 96-node (192-processor) Scali system at PC2 in Paderborn is currently the world's largest system using SCI as the interconnect technology. Since Scali are using standard workstations as nodes in the parallel systems, technology advances should ensure a continued and increasingly good price-performance ratio. By using COTS components the performance growth does not only apply to new machines, but the existing nodes of an upgraded machine can be passed on in an organization as personal desktop workstations. This adds an important option for cost reduction.

By using a standard programming interface, MPI-conforming third party applications will run on Scali systems without additional work, although optimizing for the architecture may give additional speedup. ScaMPI supports a variety of options and tools to ease the programming effort to get to correct and efficient applications. As shown by the performance measurements in Section 14.6.2, Scali systems scale well, thus enabling us to deliver very powerful systems to a low price. In this picture ScaMPI and its support modules play an important role.

Acknowledgments

Thanks to Thierry Matthey at Parallab, for making performance numbers on the Origin 2000 available, and to the service team at PC2, Paderborn, for excellent support in bringing up their 192-processor system.

References

1. David E. Culler, Andrea Dusseau, Seth C. Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing'93, Portland, Oregon*, November 1993.
2. Dolphin Interconnect Solutions. *PCI-SCI Bridge Functional Specification*, version 3.01 edition, November 1996.
3. Manolis G.H. Katevenis, Evangelos P. Markatos, and Penny Vatsolak. The Remote Enqueue Operation on Networks of Workstations. In *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing, Las Vegas, USA*, Lecture Notes in Computer Science. Springer-Verlag, February 1998.
4. G.A. Geist and V.S. Sunderam. The Evolution of the PVM Concurrent Computing System. In *Proceedings of COMPCON Spring'93*, pages 549–557, February 1993.

5. James D. Lyle. *Sbus: Information, Applications, and Experience*. Springer-Verlag, 1992. ISBN 0-387-97862-3.
6. Thierry Matthey. Personal communication, 1999.
7. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995. Version 1.1.
8. National HPCC Software Exchange – Parallel Tools Library. http://www.nhse.org/ptlib.
9. Review of Performance Analysis Tools for MPI Parallel Programs. http://www.cs.utk.edu/~browne/perftools-review/.
10. Knut Omang. Synchronization Support in I/O Adapter Based SCI Clusters. In *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing, San Antonio, Texas*, volume 1199 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, February 1997.
11. Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95, San Diego*, 1995. http://www-csag.ucsd.edu/papers/csag/external/HPVMFM-p.html.
12. Pallas GmbH. *VAMPIRtrace for Solaris x86*, 1998. Release 1.0 for VAMPIR-trace version 1.5. http://www.pallas.de.
13. PCI Local Bus Specification, Revision 2.1.
14. Scali AS. *ScaMPI Installation and User's Guide version 1.6*, 1999.
15. IEEE CS Task Force of Cluster Computing. http://www.dgs.monash.edu.au/~rajkumar/tfcc/, 1998.
16. *TotalView Multiprocessor Debugger User's Guide*, 1998. Version 3.0. http://www.etnus.com.

Part VI

Shared Memory Programming Models and Runtime Mechanisms

Due to the remote memory access capabilities of the SCI DSM, SCI clusters lend themselves readily to implement high-performance message-passing libraries. Several chapters of Part V show how remote writes can be utilized to efficiently transfer data between sender and receiver.

Yet, the SCI DSM constitutes a shared *physical* address space only, disallowing caching of remote memory contents. It is much more challenging to devise and realize shared-memory or shared-objects abstractions in this environment than it is to implement message-passing programming models. The major challenge involved is to provide a global *virtual* address space that can be conveniently and efficiently accessed by processes or threads distributed in the cluster. Solutions will also have to address caching and consistency aspects of shared memory models.

This part reports on projects either attempting to provide shared virtual memory, shared objects or shared files on top of SCI DSM, or experimenting with new runtime mechanisms facilitated by the SCI DSM, e.g., a macro data-flow execution model.

The projects and approaches reported here are widely diverse, reflecting the fact that this area bears a number of open research issues, e.g., mapping shared SCI segments to the same virtual address in distinct processes or replicating data to various nodes to improve performance while maintaining consistency. Several contributions also point out that current-generation SCI cluster hardware and device drivers are not fully adequate for shared-memory abstractions, and propose approaches to overcome the limitations.

Chapter 15 describes how the global address space abstraction is realized in the Split-C parallel language. Two implementations of the Split-C runtime system are presented, one using Active Messages as the communication layer, the other directly exploiting SCI remote memory accesses. Based on performance and functional evaluations of these implementations, their limitations are disclosed. The obstacles observed are partially due to the current

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 263-265, 1999
© Springer-Verlag Berlin Heidelberg 1999

SCI equipment, e.g., restrictions imposed on the number and location of exported and imported SCI DSM segments or high access times to exported and thus uncached local memory areas. A hybrid implementation strategy combining the shared-memory and message-passing implementations of the runtime system is proposed to overcome the drawbacks.

In Chapter 16, a comprehensive shared-memory programming interface called SMI (for Shared Memory Interface) is introduced. SMI attempts to hide peculiarities of the SCI DSM, e.g. the non-coherent memory model, from the programmer in order to keep parallelization simple. Again, some properties of the SCI hardware and driver pose problems for this approach. SMI provides a standard set of functions to allocate, distribute, and (temporarily) replicate shared memory regions, to start and distribute processes, and to allow them to synchronize. Moreover, sophisticated facilities for scheduling loop iterations on processors as well as for load balancing are available. SMI has been used in a number of application parallelizations, one of which is reported in Chapter 22.

Chapter 17 reports on a project that goes beyond these two approaches in that it builds a general global virtual memory on an SCI cluster, supporting true shared-memory programming, similar to a shared-memory multiprocessor (SMP) machine. The project combines SCI's hardware DSM and concepts of software shared virtual memory (SVM) systems to achieve the ambitious goal. The resulting system, called SCI-VM, also addresses caching of shared data over the SCI network and consistency maintenance, relying on a relaxed memory consistency model and a kernel-level virtual memory manager extending the SCI device driver and Windows NT memory management. Several shared-memory programming systems can be supported atop of SCI-VM, with two examples being described: a simple Single Program, Multiple Data (SPMD) model and a distributed, POSIX-compliant thread system. While SCI-VM is clearly experimental at the time of writing, the performance results reported so far are quite encouraging.

A file system interface to SCI DSM is described in Chapter 18. This represents a further approach for sharing memory objects, in addition providing symbolic naming and protection of shared objects. To support parallelism, the interface is also augmented with synchronization mechanisms. A prototype implementation in Linux, called SCIOS, is described, with different protocols for allocating, migrating, and replicating pages of shared files spanning the SCI cluster. A notable feature is the remote swapping facility that allows pages to be transferred to/from a remote node about 50 times faster than to/from a local disk.

Yet another approach is introduced in Chapter 19, so-called parallel CORBA objects. These parallel objects are extensions to a CORBA implementation on an SCI cluster. Thus, both distributed object-oriented programming (through regular CORBA objects) as well as parallel processing (through parallel objects) can be supported in a cluster. Parallel CORBA

objects can use message passing internally to implement the parallelism. An extended Interface Definition Language (IDL) has been developed that allows to manage data distribution among the individual objects comprising a parallel object. A runtime system, called Cobra, is presented that provides the services to execute the parallel CORBA objects, e.g., allocating all the required resources. A real-world application demonstrates the usefulness of this system.

Finally, Chapter 20 covers a somewhat "exotic" system on an SCI cluster, the Multithreaded Scheduling Environment (MuSE). MuSE is a runtime system that executes programs with appropriate structure in a macro data-flow manner on off-the-shelf hardware. That is, a computation can start execution on a cluster node when all its input data are available. The SCI DSM is shown to be a good platform for this kind of execution for two reasons: input data can be efficiently written to waiting threads and threads can be efficiently migrated to nodes destined to execute them. The chapter describes the MuSE scheduling concepts, with a focus on the MuSE work stealing algorithm and the decision whether to sequentially call (on the same node) or to concurrently spawn (on another node) a computation.

15. Shared Memory vs Message Passing on SCI: A Case Study Using Split-C

Max Ibel, Michael Schmitt, Klaus Schauser, Anurag Acharya

Department of Computer Science, University of California, Santa Barbara
email: {ibel,schmittm,schauser,acha}@cs.ucsb.edu
http://www.cs.ucsb.edu/research/sci

Non-coherent remote memory access with a simple load-store interface is one of the characteristic features of the Scalable Coherent Interface (SCI). It allows SCI to bridge the gap between efficient (and expensive) hardware implementations of true coherent shared memory and low-performance software implementations. It also allows SCI to be used as a substrate for efficient implementations of higher-level communication mechanisms. In this chapter, we evaluate the tradeoffs for using SCI to implement user-level message-passing and a global address-space scheme with shared-memory segments. We perform this evaluation in the context of Split-C, an explicitly parallel language which provides a global address-space abstraction as well as bulk transfer operations.

For the *message-passing implementation*, we use SCI to implement Active Messages and use Active Messages as the communication layer for the Split-C runtime system. For the *shared memory implementation*, we use SCI operations directly as the communication layer for Split-C. The direct implementation has the potential to provide a performance advantage by reducing copying and synchronization. However, our SCI hardware poses problems that make an efficient direct implementation very challenging. Our studies show that neither paradigm is optimal in our current setting and suggest a combination of both paradigms for increased efficiency.

15.1 Introduction

The two basic design choices for programming parallel machines are *Message-Passing* and *Shared Address Space*. SCI promises efficient implementation of both paradigms: it allows transport of messages with extremely low latency and high bandwidth, thus providing an ideal base for message-passing applications. SCI also provides direct access to remote memory using plain load-store transactions, thus providing an equally good base for shared memory applications. In this chapter, we evaluate the trade-offs between both communication schemes.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 267-280, 1999
© Springer-Verlag Berlin Heidelberg 1999

We perform this evaluation in the context of Split-C [3], an explicitly parallel language which provides a global address space abstraction as well as bulk transfer operations. We choose Split-C as it can be implemented using message-passing or shared memory. Selecting a single high-level language also allows us to focus on the differences in the communication mechanism more easily.

Our message-passing implementation maps Split-C primitives to calls to a runtime system based on a Active Messages [17] library. Our shared memory implementation called DRMA (Direct Remote Memory Access) maps Split-C primitives directly to SCI primitives.

The rest of this chapter is organized as follows. We give a brief introduction into Active Messages and Split-C in the remainder of this first section. Sections 15.2 and 15.3 discuss the message passing and the shared memory implementation respectively. Section 15.4 compares both schemes by experimental evaluation. Section 15.5 then describes a hybrid implementation which tries to combine the assets of both message passing and shared memory. Finally, we conclude in Section 15.6.

15.1.1 Introduction to Split-C

Split-C is a parallel extension of the C programming language and follows the SPMD model of computation: a single thread of computation is started on each processor. Both the parallelism and the data layout are explicitly specified by the programmer. Split-C provides a global address space in the form of distributed arrays and global pointers, and supports efficient split-phase operations (including bulk transfers) to access remote data. Arrays can be distributed in cyclical or blocked fashion, and in multiple dimensions. Global pointers are tuples of the processor number and a local address. Pointer arithmetic on such global pointers can be cyclic or blocked.

Remote data in Split-C is accessed using split-phase operations. Split-phase accesses separate the initiation of a memory operation (the request) from the response, to hide the latency of the underlying communication network. Split-C offers `put` and `store` operations for transferring data, and `get` operations for retrieving data: `put` and `get` are acknowledged transactions, while `store` is one-way. The split-phase nature of Split-C requires that the basic memory operations (`put`, `get`, and `store`) keep track of outstanding memory transactions. Every `put` or `get` request increments a counter to indicate that an operation has begun but is not yet completed. When the data has been successfully transferred, the counter is decremented. Unlike the `get` and `put` operations, a `store` operation is one-way and uses two counters: one is incremented on the sender when data is sent, and the other is incremented on the destination when the data is received. A node can issue explicit `sync` statements to guarantee that all outstanding requests have been finished. The synchronous `read` and `write` operations are shortcuts for `get` or `put` followed by a `sync`. Control flow in Split-C is augmented with barriers

and atomic operations. To allow optimizations for parallel programs, Split-C also offers library routines for large transfers (`bulk_put` and `bulk_get`) and scatter-gather operations.

Split-C implementations have been built on top of Active Messages on many computational platforms, including the TMC CM-5, Intel Paragon, Meiko CS-2, IBM SP-2, and networks of workstations [11]. On a few machines like the T3D, Split-C has been built directly on the hardware [1].

15.1.2 Introduction to Active Messages

Active Messages is a low-latency communication mechanism. Each active message contains the address of a handler function which is executed on the receiving processor upon arrival of the message. Message handlers are intended to be short and execute quickly. In particular, message handlers are not allowed to suspend. To eliminate complicated buffer management and simplify deadlock considerations, Active Messages divides handlers into two classes: requests and replies. This is similar to the request/reply protocol underlying SCI. An active-message request can send a message to an arbitrary processor; when it arrives at its destination, the specified request handler is invoked. Request handlers may answer by sending a single reply message, while reply handlers are prohibited from additional communication. Under the Active Messages model, messages travel from user space (the send instruction) directly to user space (the message handler), avoiding any form of buffer management and synchronization usually encountered in the traditional send & receive model. As a result, Active Messages can achieve an order of magnitude performance improvement over more traditional communication mechanisms.

Although quite primitive, Active Messages has become an important communication layer because of its efficiency. Active Messages has been implemented on many different hardware platforms, including the Meiko CS-2 [13], Cray T3D [1], as well as clusters of workstations connected by FDDI [12], ATM [16] and Myrinet [5]. The small overhead and low latency facilitates building more complicated communication layers [15], makes it a desirable target for high-level language compilers [4], like Split-C.

15.2 Message-Passing Implementation

We first describe our implementation of Active Messages on top of SCI. Next, we describe the implementation of Split-C on top of Active Messages. Further information on our message-passing implementation can be found in [7, 8, 9].

15.2.1 Active Messages on Top of SCI

There are several issues that an implementation of Active Messages must deal with. We discuss protection, reliability, notification, and synchronization.

To provide **protection**, we must ensure that an unauthorized process cannot interfere with another process on any node in the network. SCI leverages the virtual memory management provided by the operating system to achieve protection. The SCI device driver controls the mapping of remote memory segments into a process' address space. Processes that do not share a mapped memory segment cannot communicate.

To provide **reliability**, we must ensure that each message is delivered to the destination once and only once. This is still a major research issue for clusters of workstations which usually are connected by an unreliable network which may drop packets (e.g. ATM). Fortunately, the SCI standard guarantees that all messages are delivered and are delivered at most once. However, SCI provides no guarantee concerning message ordering. Because of the simple request-reply nature of Active Messages and the fact that messages are sent in atomic network transactions, this does not pose a problem though.

For proper **notification**, we must ensure that the receiving node is notified about incoming communication so that it can take appropriate action. In the case of Active Messages, this means that the processor is notified upon the arrival of a message so that it can invoke the message handler. There are two standard ways of notifying the main processor: using interrupts or using polling. Since interrupts are far too costly on most architectures, the common solution is to have the processor check (poll) at regular intervals whether a new message has arrived.

Polling also simplifies any required **synchronization** between the ongoing computation and incoming messages. The processor just does not poll in critical sections, which ensures that incoming messages do not interfere in unexpected ways with the running computation. However, in a non-dedicated workstation polling wastes CPU time available to other processors.[1]

The ideal abstract data type for dealing with notification and synchronization in this environment is a queue. This simple insight underlaid the experience presented in [13] and was formalized in [2], where it was named the *Remote Queue* abstraction. Sending processors just enqueue their message on the remote queue. During a poll, the receiving processor just checks whether something has been enqueued. If so, it removes the message from the queue and processes it. This remote queue abstraction can be built easily on traditional message-passing network interfaces, because they have a single point of entry which essentially acts as a queue. It is the receiving processor's responsibility to pull the messages from the network interface and process them.

[1] In non-dedicated environments though, interrupts become more attractive since they can achieve co-scheduling. Further information about an alternative message-passing implementation suitable for non-dedicated environments by using a combination of polling and interrupts can be found in [14].

A Simple Remote Queue Implementation

In our current implementation each processor establishes a queue for communication with every other processor. This setup is shown in Figure 15.1.

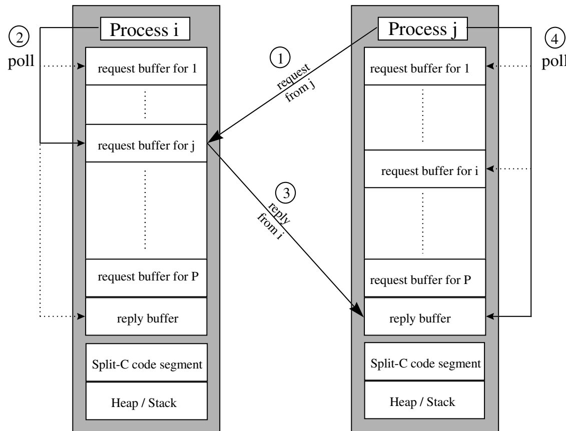


Fig. 15.1. The implementation of Active Messages on top of SCI.

To check whether a message has arrived, a processor checks *all* of the incoming request buffers. If any one contains a message, the receiving processor extracts the message, processes it, and sends a reply back. The poll also checks whether any of the outstanding requests have been replied to yet and extracts the reply and processes it. A processor that wants to send a request to another processor with a full request queue polls the network until outstanding requests have been replied to and free entries in the queue are available. To ensure that no deadlocks can occur, a processor that is waiting for a reply constantly polls the reply buffer as well as all of the incoming request buffers. Under our scheme, every request has to send a reply, so that the requester knows when it can reuse the request buffer. If an active message request handler does not send a reply (for example in the case of the one way Split-C `store` operation) we automatically insert a null reply message.

For our implementation we only use (non-blocking) SCI store operations. On each processor we allocate a shared segment which can be accessed by all other processors. Each processor reserves on its shared segments one reply buffer as well as P incoming request buffers, one for each of its P peers. Each of the message buffers is 64 bytes long and can be filled with a single 64-byte block move instruction.

In addition, each processor dedicates for each peer a full page (4 kByte) in the memory segment it exports to bulk transfers. To perform a bulk transfer, we first use a tight inner loop of 64-byte block moves to store the data in the

remote page. Then we send a single active-message into the request buffer indicating that the bulk data was deposited. The active-message handler then moves the data into the correct location.

15.2.2 Split-C on Top of Active Messages

Implementing Split-C on top of Active Messages is straightforward. The remote memory operations have to be mapped onto an equivalent set of active-messages and the corresponding handlers.

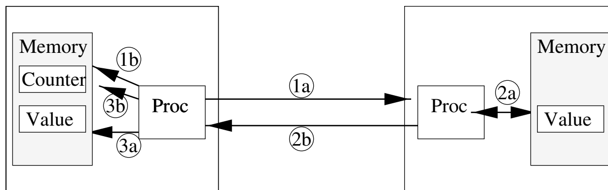


Fig. 15.2. Steps required for a Split-C `get` operation based on Active Messages.

Since `get`, `put`, and `store` all work in a similar fashion, we explain only the implementation of a `get` in detail. The `get` operation applied to a global address is sketched in Figure 15.2. It consists of the following three steps:

1. The requesting processor
   a) examines the global address, extracts the processor number and sends the `get` request to the remote processor and
   b) increments the counter tracking outstanding requests.
2. The remote processor receives the request message,
   a) reads the requested data, and
   b) sends it back.
3. The requesting processor receives the reply,
   a) stores the data, and
   b) decrements the counter.

Step 1 corresponds to sending an active-message, while Steps 2 and 3 correspond to active-message handlers (servicing the request or reply message, respectively).

A `put` works in a very similar fashion, except that the request (1a) sends the data to be stored on the remote side and the reply (2b) does not carry any data but is only used to decrement the counter. The `store` operations are inherently one way. The originating node increments its store counter for each `store`, and the receiver also increments a local counter for every received `store`. There are two possible synchronization mechanisms: When the receiver knows how many `store` operations to expect, it can just wait

for a specified counter value. If the number of stores that have been issued is unknown to the receiver, the `all_store_sync` primitive performs a global and thus more expensive synchronization, similar to a barrier, to ensure all stores across the cluster have been completed.

The synchronous `read` and `write` operations are implemented using a `get/put` operation followed immediately by a `sync`. A `sync` itself just waits in a tight loop for all or specific outstanding `get`s or `put`s to finish.

15.3 Shared Memory Implementation

SCI's shared address space support provides a global address space in hardware and enables the optimization of reduced copying and synchronization overhead compared to a standard message-passing implementation. For example, the `get` can fetch the value from the remote memory using a plain shared memory load operation; this leads to performance improvement, mainly since additional copying to and from receive buffers is avoided and since the CPU on the receiver side is not involved in these communication events. In this section, we describe the implementation of Split-C directly on top of SCI (we refer to this as the DRMA—Direct Remote Memory Access—implementation). Further information about our shared memory implementation can be found in [10].

15.3.1 Split-C on Top of SCI

For now, we assume that the complete application memory (heap and stack) of every node is exported to every other node. We examine the `get`, `put`, `store`, `read`, `write` and `sync` as well as bulk transfer primitives and their mappings to shared address space operations.

In contrast to the Active Messages implementation of `get` which requires two 64-byte SCI stores, the DRMA implementation only needs one SCI load operation[2] to fetch the requested data and does not have to involve the processor on the node which holds the data. Similarly, the `put` maps to a simple SCI store. The `store` operation is equivalent to `put` since all primitives in the DRMA implementation are inherently one-way. The `read` operation is equivalent to `get` since the SCI load operation is blocking. The `write` operation is similar to `put` but since the SCI `store` operation is not blocking, it requires an additional memory barrier to ensure the completion of the `store`. A major advantage of DRMA is that `sync` can be implemented as a simple memory barrier if we have outstanding `put` or store operations, otherwise as a NOP. This significantly decreases the amount of necessary synchronizations and thus better decouples the control flow across node boundaries.

[2] After a table lookup to determine the mapping of the Split-C global pointer to the requested data in the pool of imported segments.

Bulk operations can also be simplified. The semantics of Split-C asserts that the destination of a message is specified by the sender, such that no queuing needs to be performed by the receiver. This allows the DRMA implementation to write incoming bulk messages directly to the destination address, thus avoiding an extra level of data copying. In addition, the receiver need not be notified of incoming messages.

15.4 Experimental Evaluation

We measured and compared the performance of both the message-passing (Active Messages) and the shared memory (DRMA) implementation on our SCI platform using a set of micro-benchmarks and application benchmarks. For this study, we used four Ultra-30 workstations with 250 MHz CPUs, 128 MByte RAM, and PCI-based Dolphin network adapters using the LC-2 link controller [6]. We used both a ring and switch topology but found for our set of benchmarks that the switch does not improve throughput and only increases latency. All of the measurements which are presented here were therefore executed on a four-node SCI ringlet.

15.4.1 Micro-benchmarks

The simpler implementation of the Split-C primitives in the DRMA version compared to the Active Messages version results in an improved latency for the Split-C communication primitives. Table 15.1 shows the overhead of the `get` and `put` primitives and the latency of the `read` and `write` primitives for the Active Messages-based and for the DRMA-based runtime system. We are assuming 8-byte data items in the DRMA case. Also, for message-passing, we assume that the receiver is always polling and can immediately respond to incoming messages, which is the best case. The overhead for the Active Messages implementation is the sum of sender and receiver overhead.

	`get`	`put`	`read`	`write`
Active Messages	$5.8 \mu s$	$5.8 \mu s$	$13.5 \mu s$	$13.5 \mu s$
DRMA	$5.6 \mu s$	$2.9 \mu s$	$5.6 \mu s$	$11.5 \mu s$

Table 15.1. Overheads (`get`, `put`) and latencies (`read`, `write`) for Split-C primitives.

As expected, DRMA outperforms Active Messages for all cases by avoiding buffering, by not involving the processor on the node containing the accessed data and by reducing the amount of synchronization necessary across node boundaries.

Figure 15.3 shows the latencies of the Split-C operations `bulk_get` and `bulk_put` for the Active Messages based and for the DRMA-based runtime system.

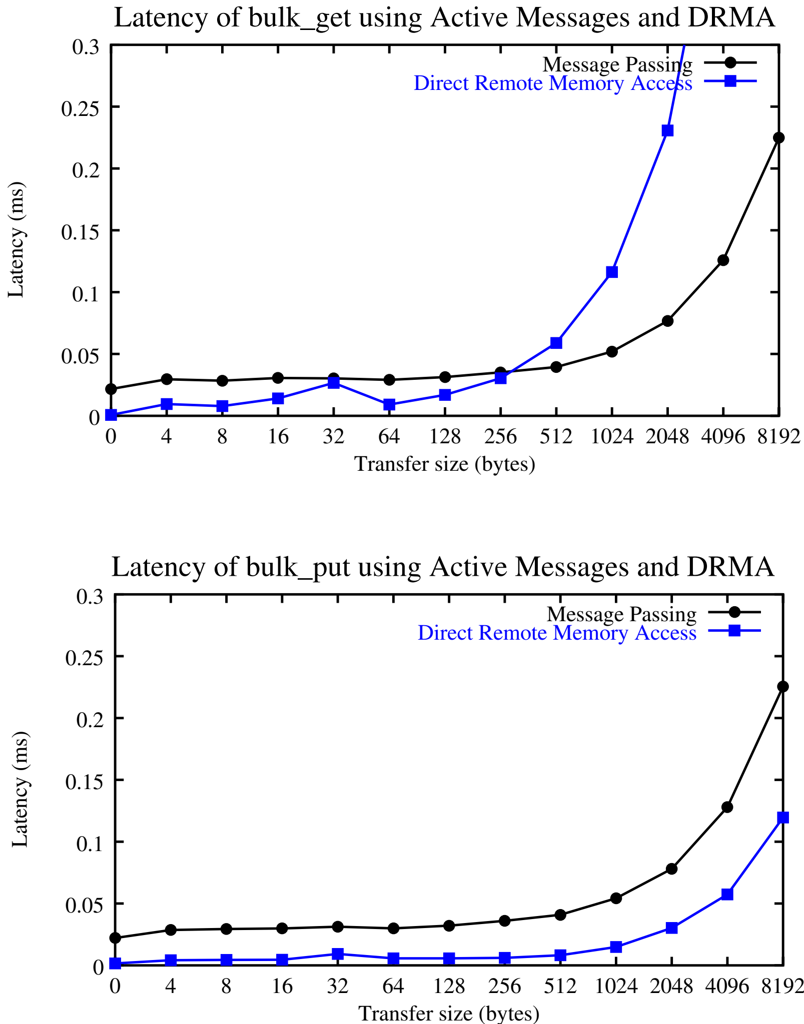


Fig. 15.3. Comparison of Split-C `bulk_get` and `bulk_put` latencies using Active Messages and DRMA.

For `bulk_put`, DRMA performs consistently better than Active Messages as expected. For `bulk_get`, we see that the Active Messages-based version performs worse for small data sizes and better for larger sizes, with a break-even point at about 256 bytes. The reason is that the Active Messages version is based on stores: the initiator of a `get` asks the owner of the data to write

it into the initiators memory using stores. For DRMA, the initiator directly accesses the memory using load instructions, which are much slower (since blocking) than stores.

We measured a sustained SCI store bandwidth of 66.6 MByte/s and a sustained SCI load bandwidth of 9.4 MByte/s on our cluster. In our implementation, we therefore use the direct `bulk_get` implementation only for data sizes of up to 256 bytes and use message passing for larger data sizes. This can be seen as a hybrid implementation that utilizes either the message-passing or the shared address space approach whenever appropriate. As section 15.5 shows, there are more reasons to switch between the paradigms.

15.4.2 Application Benchmarks

We also ran several parallel applications to confirm our results. The measurements are summarized in Table 15.2.

	Active Messages	DRMA
heat diffusion		
`bulk_put`	$1.02ms$	$1.05ms$
`write`	$4.28ms$	$4.21ms$
`put`	$2.69ms$	$1.55ms$
`read`	$4.35ms$	$2.88ms$
`get`	$2.74ms$	$2.88ms$
FFT (2M)	$7.27s$	$9.58s$
computation	$1.56s$	$5.15s$
communication	$5.71s$	$4.43s$
Barnes-Hut (32K)	$40.7s$	$60.3s$

Table 15.2. Application benchmarks for Split-C implementations.

We used several versions of a heat-diffusion kernel (grid size 128*16), each of these employing a different Split-C primitive (`bulk_put`, `write`, `put`, `read`, `get`) for the communication phases. The `bulk_put` version with Active Messages takes $1.02ms$ per iteration (computation + communication), whereas the DRMA version takes $1.05ms$. To see that the DRMA implementation is somewhat slower is confusing at first. However, we measured the communication and computation time independently and found that computation accounts for more than 90% of the execution time. While the communication time using DRMA is cut by about 50%, the computation time increases about 10% and thus causes a net increase in execution time. The reason for this is that memory imported from another node is never cached—remote references always result in a network access. Especially, *local memory accesses to the pinned-down exported memory segments perform worse than local memory accesses to not exported segments.* Exporting of a substantial part of the application memory can therefore slow down the computation significantly and

thus nullify the increased communication performance with DRMA. This adverse effect is lessened in the other versions (especially `put` and `read`) since the communication phases are here more dominant in the execution time.

We also ran a fine-grained FFT kernel with 2M data points. This application is highly optimized for locality and we can see that although DRMA improves on communication time, the local computations are slowed down by a factor of more than three. Finally, we ran ten time steps of a Barnes-Hut simulation with 32K bodies. Like FFT, Barnes-Hut performance drops significantly when using DRMA, due to slower local computation and slower memory accesses.

The non-cacheable nature of local exported memory segments is an artifact of our current SCI hardware and device drivers. When this limitation is removed, we believe that DRMA will be usable for a wider range of applications than it currently is.

15.5 Hybrid Implementation

The Dolphin SCI adapter card is attached to the PCI I/O bus and therefore cannot access an arbitrary virtual address in a remote process (as may be required by the Split-C semantics). Remote accesses are limited to previously allocated remote memory segments, each of a maximum size of 512 kByte. It is not possible for large programs to allocate all of the globally addressable data structures into the remote segments.[3] Thus, it is necessary for the remote processor to service some requests, since it is the only entity capable of doing the address translation. In this section we describe a basic framework and mapping strategies for a hybrid implementation which utilizes both the DRMA and the Active Messages runtime system.

15.5.1 Basic Framework

The basic framework of our hybrid implementation services remote requests by message-passing if the requested data is not in a currently mapped remote segment. If the current strategy permits (see Section 15.5.2 for a discussion of possible mapping strategies), we re-map exported segments to incorporate the requested data in the current working set of the global address space. The procedure is as follows:

1. On the requesting node:
   a) If the requested data is on a currently mapped remote segment, fulfill the request with DRMA; if not, fulfill the request with message-passing.

[3] The limit for the total application memory (on all nodes) is with our current SCI cards 40 MByte, since we cannot map more than 80 segments at any time.

   b) If the current mapping strategy permits, send a mapping request as a special active message to the node that holds the requested data; if not, return successfully.
2. On the node holding the requested data:
   a) Unmap the existing exported segment according to the current mapping strategy.
   b) Map the segment with the requested data (using the `mmap()` interface).
   c) As long as (a) and (b) have not finished, deliver the requested data (on the not yet exported memory segment) with the message-passing interface.

Note, that we do not require the more expensive[4] `sci_map_shm()` interface for steps 2(a) and 2(b). We can avoid this by employing `sci_map_shm()` only during startup to create a pool containing the maximum possible number of exportable segments and using only `mmap()` and `memcpy()` during re-mapping to change the virtual address translation and move data between the pinned down pool of exported segments and the heap.

However, the cost for re-mapping memory segments is still relatively high ($110\mu s$ in the best case) since it requires in addition to the `mmap()` and `memcpy()` a broadcast of the new mapping to all nodes that import that segment. Therefore, an efficient re-mapping strategy is important.

15.5.2 Mapping Strategies

We propose three different strategies to co-employ shared address space and message-passing in the design of a global address space:

– *Greedy*: Whenever a segment is not yet mapped, we perform the memory access using message-passing, but also make sure that the requested segment is mapped as soon as possible. Evicted segments are chosen randomly.
– *Adaptive*: We keep counters on memory segment hits and only map a new segment after a configurable number of requests via message-passing. The segments with the fewest number of hits since the last eviction decision can be expelled.
– *Sampling*: We monitor the program using only message-passing for a fixed time period (sample phase). After that period, we map only the most profitable memory segments and use this static memory layout for another time period (production period), after which we repeat the process.

[4] We measured the time for establishing a memory mapping using PCI-SCI to be in the range of several *hundred milliseconds*.

15.6 Conclusions

In our study, we have analyzed two different implementations of Split-C on an SCI cluster. In the message-passing implementation the Split-C primitives map to an Active Messages-based runtime system, whereas in the shared memory implementation (DRMA) they map directly to corresponding SCI primitives.

We have shown that SCI is able to provide an efficient global address space substrate as well as good message-passing performance. However, while the raw communication performance of SCI in the DRMA implementation is impressive, our experiments showed that our current hardware and drivers impose constraints that limit the utility of SCI when used for a direct shared address space implementation.

A major obstacle is the limited number of memory segments that can be imported. While the on-board address translation table of our Dolphin SCI cards allows up to 8192 entries, we have not been able to import more than 80 segments at any time. Furthermore, on systems without an I/O-side MMU, each imported memory segment needs to start on a physical 512 kByte-boundary, which limits the number of exportable memory segments. To some extent, this can be alleviated by choosing large memory segments (up to 512 kByte), or by dynamically mapping/unmapping memory segments between nodes. The high cost for map/unmap can be mitigated by a hybrid implementation that switches to Active Messages to hide the segment mapping latency.

Another problem is the larger memory access time of local (but exported) data: We observed that while normal cache lines can be fetched from the L1 and L2 cache in $12ns$ and $40ns$, respectively, every access of local memory that has been exported to other node takes roughly $60ns$. This means that only applications that have a high communication-to-computation ratio should map memory remotely. This complicates the design of the runtime system and the compiler and may preclude the use of direct memory accesses for many programs with a broad communication footprint.

Nevertheless, a direct remote memory implementation of Split-C allows remote data accesses without any processor involvement on the remote processor, unlike a message-passing implementation. We believe that an architectural improvement of the SCI adapter cards together with an optimized hybrid Split-C implementation as outlined in Section 15.5 can effectively combine the assets of both message-passing and shared memory for our SCI platform.

References

1. R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY T3D: A compiler perspective. In *International Symposium on Computer Architecture*, June 1995.

2. E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *7th Annual Symposium on Parallel Algorithms and Architectures*, July 1995.
3. D. E. Culler, A. Dusseau, S. C. Golstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing*, November 1993.
4. D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18, July 1993.
5. D. E. Culler, L. T. Liu, R. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, 1996.
6. Dolphin. *PCI-SCI Adapter Programming Specification*. Dolphin Interconnect Solutions Inc., November 1997.
7. M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. Implementing Active Messages and Split-C for SCI Clusters and Some Architectural Implications. In *Sixth International Workshop on SCI-based Low-cost/High-performance Computing*, September 1996.
8. M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. High-Performance Cluster Computing Using Scalable Coherent Interface. In *Seventh International Workshop on SCI-based Low-cost/High-performance Computing*, March 1997.
9. M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. High-Performance Cluster Computing Using SCI. In *Proc. of Hot Interconnects V*, August 1997.
10. M. Ibel, M. Schmitt, K. E. Schauser, and A. Acharya. An Efficient Global Address Space Model with SCI. In *Proceedings of SCI Europe '98*, September 1998.
11. A. Krishnamurthy, K. E. Schauser, C. J. Scheiman, R. Y. Wang, D. E. Culler, and K. Yelick. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.
12. R. P. Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Proc. of Hot Interconnects II*, August 1994.
13. K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *9th International Parallel Processing Symposium*, April 1995.
14. M. Schmitt, M. Ibel, A. Acharya, and K. E. Schauser. Adaptive Receiver Notification for Non-Dedicated Workstation Clusters. In *Proc. of the 1998 Int'l Conference on Parallel Architectures and Compilation Techniques*, October 1998.
15. L. W. Tucker and A. Mainwaring. CMMD: Active messages on the CM-5. *Parallel Computing*, 20(4), April 1994.
16. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. Symposium on Operating Systems Principles*, 1995.
17. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

16. A Shared Memory Programming Interface for SCI Clusters

Marcus Dormanns, Karsten Scholtyssik, Thomas Bemmerl

RWTH Aachen, Lehrstuhl für Betriebssysteme, Kopernikusstr. 16,
D-52056 Aachen, Germany
email: contact@lfbs.rwth-aachen.de
http://www.lfbs.rwth-aachen.de/

16.1 Introduction

Maybe the most noticeable and distinctive feature of SCI is its shared memory capability, i.e., the transparent access to segments of shared memory, exported by remote compute nodes. However, until now this is only rarely employed for application parallelization. Instead, message passing on top of SCI is often used (see, e.g., Chapters 13 and 14) for several reasons:

- Performance: Message passing programming requires to explicitly code communication operations. Doing so, it is often simpler or at least more natural to take care of performance aspects [6].
- Software legacy: A large number of already existing parallelized applications are based on message passing, typically employing MPI or PVM. This is a good motivation for standard message passing programming interfaces on top of SCI.
- SCI peculiarities: The type of shared memory that is offered by current implementations of I/O bus-based SCI adapters comes with some peculiarities (see [2] and Chapter 3):
  - The lack of a cache coherency layer forces to completely switch off caching of remote memory to keep the memory model simple.
  - The probability of data transmission errors in certain situations.

But there are also good reasons for shared memory programming, which highly justify to explore shared memory application parallelization on SCI clusters:

- Simplicity: Shared memory parallelization is commonly accepted as being simpler than dealing with message passing. This applies especially to the situation when an already existing sequential application is subject to parallelization and the parallel program cannot be developed from scratch.
- Scalability of the parallelization process itself: Relying on the shared data paradigm, it is possible to parallelize one time-critical module after another. This provides the ability to arbitrarily scale the amount of work (and therefore money) spent in the parallelization process with respect to the expected performance benefit.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 281-290, 1999
© Springer-Verlag Berlin Heidelberg 1999

- Computer architecture trends: While most parallel architectures of the past followed the distributed memory approach combined with message passing programming, several recently introduced machines (e.g., Sequent NUMA-Q and SGI Origin; see [3] for an overview) follow the NUMA shared memory approach, as SCI does.

In this chapter, a programming interface that enables a convenient and efficient shared memory parallelization on SCI clusters, called *SMI* (for *Shared Memory Interface*), is introduced. It has been developed with special emphasis on hiding functional and performance peculiarities of SCI from the programmer to allow a simple and efficient parallelization process. It offers standard shared memory programming services, like allocation of shared memory regions and synchronization, as well as sophisticated parallelization support, e.g., loop scheduling.

SMI is implemented as a library on top of basic SCI functions that are provided by device drivers (e.g., creating and mapping cluster-wide segments of shared memory; see Chapter 3). Bindings for C/C++ as well as for Fortran 90 (or Fortran 77 with DEC/Cray pointer extensions) are provided. Supported operating systems are Unix (e.g., Solaris and Linux) as well as Windows NT which is especially popular on PC-based clusters. The applicability of SMI is not limited to SCI clusters. It may be used on all systems that provide shared memory segments in some way. Examples that have already been incorporated are symmetrical multiprocessor architectures (workstations, PCs, and a HP/Convex SPP) and LAN-interconnected clusters with a software shared virtual memory (SVM) system [7].

In the following sections, the properties of current SCI clusters that influence shared memory application programming are analyzed and the application programming interface is sketched.

16.2 Platform Properties: System Image and Memory Model

16.2.1 System Image and Operational Model

In distinction to a dedicated parallel system like those from Sequent or SGI, a cluster is not operated by a single system-wide operating system, but by individual autonomous operating system instances on the compute nodes. Under these circumstances, there is no standard procedure to share an entire process' address space among the cluster compute nodes, nor to spread and schedule several threads of a single process on different nodes. The reasons are that neither the standard virtual memory managers nor the schedulers are coupled. This prohibits use of the well-known thread-based shared memory programming model [4], if one is not willing (for good reasons) to accept

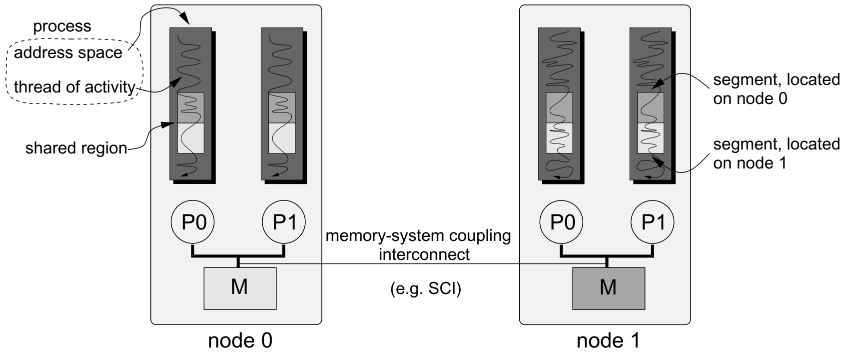


Fig. 16.1. The operational model of SMI

considerable adaptations of the given platform (i.e., operating system and SCI device driver). Nevertheless, it is possible, as described in Chapter 17.

The alternative, which is employed in SMI, is to base a programming model on individual globally shared segments that are shared among autonomous processes (i.e., mapped into their virtual address spaces) that are executed on the different cluster compute nodes. For multiprocessor compute nodes, a user can freely choose between using multiple threads within a single process per compute node or setting up multiple processes per compute node, each with a single thread, avoiding to mix different shared data programming methodologies. Figure 16.1 illustrates SMI's operational model.

16.2.2 Memory Model

The memory model of a NUMA shared memory parallel system deals with two aspects:

– performance and
– coherency.

While performance aspects have already been addressed in earlier chapters in depth, coherency (which is not limited to cache coherency) is an essential functional aspect. While remote memory cannot be cached due to the inability of implementing SCI's cache coherency layer on an I/O-bus network adapter card, it is possible to cache globally shared memory at the compute node on which it is physically located. Note that switching off caching alone does not re-establish data coherency automatically. Prefetched reading (by the SCI adapters) as well as write buffers (on the SCI adapters as well as on the processors) generally prevent data coherency. This applies already to an individual symmetric multiprocessor [1].

It is essential to hide a non-coherent memory model from an application programmer to maintain simplicity. The common way to do so is to invali-

date read buffers and/or to flush write buffers appropriately at synchronization points. This is also done by SMI and eventually results in a memory consistency model that is known as *release consistency*.

16.3 User Front-End

On Windows NT clusters, several features which are essential for parallel processing and considered to be standard features on Unix clusters, are missing:

- invocation of the execution of a process on a remote compute node, and
- I/O redirection to other compute nodes.

For this purpose, a graphical front-end together with a remote execution service have been developed. Both are not limited to SMI programs, but also used to start, e.g., *SVM* (*Shared Virtual Memory*) parallel applications [7]. The main features are:

- Simple interactive specification of the run-time configuration, e.g.,
  - the number of processes,
  - the machines to execute the processes on,
  - the user account to execute the processes in, and
  - the common command line parameters.
- A remote execution service to start the processes of a specified configuration on the respective compute nodes within the specified user environment.
- Redirection of input and output (standard streams as well as error streams) into the front-end, using a separate window for each process.
- Resource allocation and management features, e.g.,
  - limiting the set of compute nodes that can be selected (e.g., select only those with a SCI adapter),
  - locking of compute nodes to prevent undesirable interferences of different applications.

Figure 16.2 shows a screen-shot of this front-end.

16.4 The Application Programmer's Interface

The following subsections describe the major functions of the Shared Memory Interface.

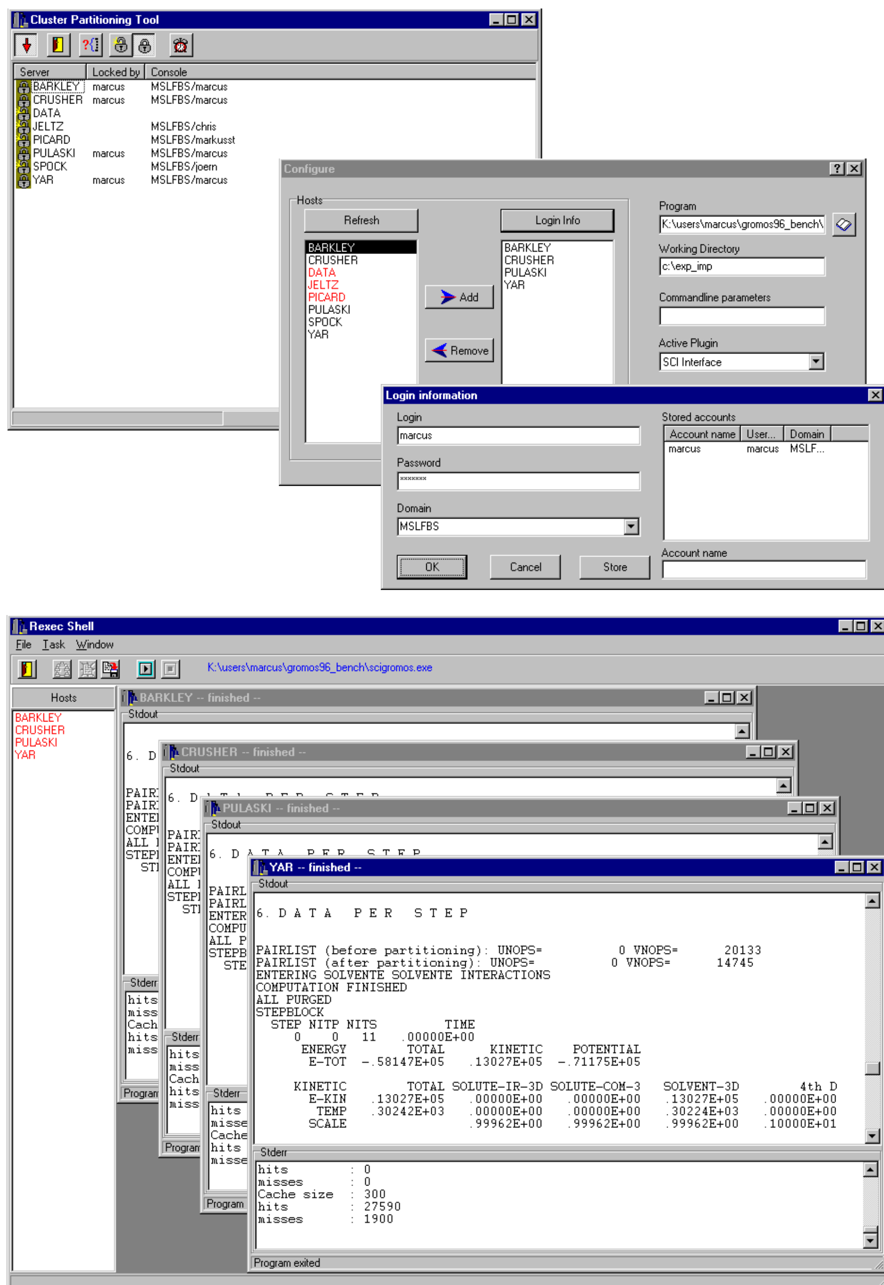


Fig. 16.2. Screen-shots of the frontend, used to manage parallel applications (top: specification of configuration and resource management, bottom: I/O redirection).

16.4.1 Initialization and Execution Environment

Before any cooperation and communication of the individual processes of a parallel application is possible, it is necessary to call the initialization function `SMI_Init`. This function has to be called *collectively*. This means that it represents a global synchronization point. It is not necessary that all processes enter the function at the same time, but no process will return from such a function until the last process entered it.

During this initialization, some globally shared memory segments are allocated for internal purposes, e.g., to store synchronization variables. After initialization, each process can request a couple of parameters regarding the run-time configuration with a call to the respective function, e.g.,

- the total number of processes, P,
- a unique process rank between 0 and $P-1$,
- the number of (multiprocessor) machines employed, M,
- a unique rank of the machine that a process is executing on,
- the ranks of the machines that other processes execute on.

Machine ranks and even more information about which processes execute on the same (multiprocessor) compute node are of interest to enhance data locality and therefore to increase application performance within a NUMA environment. To simplify this, SMI ensures that processes which execute on the same compute node possess consecutive process ranks.

16.4.2 Memory Management

The building blocks of applications which have been parallelized with SMI, are *shared memory regions*. A shared memory region is a consecutively addressable global memory section which is mapped into each process' virtual address space. To enable the usage of pointers and their exchange among different processes, SMI maps shared memory regions to identical addresses in each process' virtual address space. This is an essential precondition for dynamically assembled data structures like lists, trees, etc.

To account for the NUMA performance characteristic, each shared memory region can be assembled out of several segments which can be physically located on different cluster nodes. A programmer has the possibility to choose between three assembly policies: `UNDIVIDED`, `BLOCKED`, and `CUSTOMIZED`. `UNDIVIDED` states the simplest case in which the entire region consists just of a single segment that is allocated on a specified compute node. In a `BLOCKED` distribution, each compute node contributes a physically local segment to the entire region. All segments are mapped consecutively to establish a single undivided shared memory region. In this case, the size of each segment is proportional to the number of processes, residing on the respective compute node. With the `CUSTOMIZED` distribution policy, a programmer can specify an arbitrary concatenation of segments.

The collective function call to establish a shared memory region is:

```
SMI_Create_shreg(int TotalSize,
                 int DistPolicy, int* DistParams,
                 int* RegionId, void** Address)
```

The returned RegionId is used afterwards to refer to the corresponding region.

For several applications, a flat piece of globally shared memory might be sufficient, e.g., to host a large array of some type. But for other applications, dynamic memory allocation inside a shared memory region might be important. SMI provides the possibility to install a memory manager for a shared memory region with a collective call:

```
SMI_Init_shregMMU(int RegionId)
```

The memory manager itself allocates all data structures within that region (to keep track of the free and allocated memory portions) to allow all application processes to dynamically allocate memory within that region. This can be done with the functions:

```
SMI_Imalloc(int Size, int RegionId, void** Address)
```

and

```
SMI_Cmalloc(int Size, int RegionId, void** Address)
```

The SMI_Imalloc function can be called by individual processes. Atomicity and therefore correctness of such operations is ensured by SMI. In contrast, SMI_Cmalloc is a collective function. Its only difference to SMI_Imalloc is that upon return, all processes know the start address of the allocated piece of memory while with SMI_Imalloc only a single process does. Corresponding to these functions, SMI_{I|C}free are used to release dynamically allocated memory.

The fact that the remote fraction of a globally shared memory region cannot be cached frequently accounts for serious performance degradations (see Chapter 17). Observing that it is not necessary for a parallel application to really share a data structure all the time, SMI provides services to temporarily replicate a shared region by the function:

```
SMI_Switch_to_replication(int RegionId, ...)
```

and to switch back to sharing with:

```
SMI_Swich_to_sharing(int RegionId, int CombineMode, ...)
```

An important property of this functionality is that the replicated region resides at the same address of the process' virtual address space as the shared region did before. Therefore, all pointers remain valid. Although a region is

replicated, it can be meaningful to allow local modifications by the processes, as long as it is possible to combine the replications to a common consistent view afterwards. This is for example the case if the region contains an array that is used for accumulation operations. The common consistent view can then be re-established by summing all the replicated instances. Several such operators are provided and can be specified by the parameter `CombineMode` (e.g., *max* and *add* for different data types).

16.4.3 Synchronization

For the purpose of process synchronization, *mutexes*, *barriers*, and *progress counters* are provided. All those synchronization primitives ensure an easy-to-use memory model for the application programmer by suitable invalidation/flushing of read/write buffers. The resulting consistency model is commonly referred to as release consistency [1].

For the use of mutexes, the functions `SMI_Mutex_init`, `SMI_Mutex_lock`, `SMI_Mutex_trylock`, `SMI_Mutex_unlock`, and `SMI_Mutex_destroy` are provided, implementing the common semantics. An application-wide barrier is performed by calling `SMI_Barrier`. A progress counter denotes a set of counter variables, one for each process. Each process can increment its own counter (`SMI_Increment_PC`) and wait until the counters of a single other process or all other processes reach a certain value (`SMI_Wait_individual_PC` or `SMI_Wait_collective_PC`).

16.4.4 Loop Scheduling

Typically, loops are the major sources of compute time demands in algorithms. Therefore, most parallelization approaches try to split a loop's index range into independent subsets that are executed by different processes. A major requirement of such splittings is that the computational loads induced by the subsets are evenly balanced. Determining such a splitting can be quite complicated. The computational demands of individual loop iterations may vary with the loop indices and may be unknown in advance. In presence of a NUMA performance characteristic, the computational demand may further depend on the specific node that a loop iteration is being processed on. Depending on the shared data that is required to process a certain loop iteration and the physical location of that data, there will be well-suited and ill-suited processes to schedule the iteration upon.

To free the programmer from the burden of determining an efficient loop splitting, advanced loop scheduling algorithms have been incorporated into SMI [8]. The approach is to install one work queue for each process. The total index range is initially split among the work queues such that each process can request indices without interference with others. During loop processing, the indices are granted in small chunks to the requesting processes, decreasing in

size with increasing progress. In case that the local work queue of one process runs out of work, indices are automatically transferred from highly loaded work queues to the empty one. Therefore, load is dynamically balanced.

SMI's loop scheduling facilities extend this strategy to NUMA clusters in that the physical distribution of data (i.e., shared regions) is considered for the initial index assignment to work queues as well as for the dynamic load balancing process.

A typical code fragment that employs SMI's loop scheduling services to parallelize a loop is sketched below:

```
SMI_Loop_init(&LoopId, TotalStartIdx, TotalEndIdx,
               SMI_PART_ADAPTED_BLOCKED);
do
{
    SMI_Get_iterations(LoopId, &LoopStatus,
                       &ChunkStartIdx, &ChunkEndIdx);
    for (i=ChunkStartIdx; i<=ChunkEndIdx; i++)
    {
        /* process index i */
    }
} while (LoopStatus != SMI_LOOP_READY);
```

Several further functions allow to specify the load balancing strategy in more detail.

16.5 Conclusions

SMI has been developed for shared memory application parallelization on all types of parallel machines that provide shared memory in some form, but with special emphasis on SCI-interconnected cluster systems. SMI has already been used in several parallelization efforts:

- the GROMOS96 molecular dynamics simulation code (Chapter 22),
- a module of an airline flight scheduling system [5],
- simulation of control problems with Matlab/Simulink, and
- simulation of room acoustics [9].

SMI started as a small library with just a few functions and grew over time to cope with the demands raised by these projects. It now comprises about 50 functions.

In our projects, we found that shared memory parallelization on SCI clusters is a meaningful approach regarding both, the required effort for the parallelization process itself and the performance of the resulting parallel application. SMI has proved to offer a suitable application programmer's interface for this purpose.

References

1. S.V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, Vol. 26, No. 12, pages 66–76, Dec. 1996.
2. R. Butenuth and H.-U. Heiß. Shared-Memory Programming on PC-based SCI Clusters. *Proc. SCI Europe '98*, held as a stream of EMMSEC (European Multimedia, Microprocessor Systems and Electronic Commerce Conference and Exposition), Sept. 1998.
3. D. Culler, J.P. Singh, and A. Gupta. *Modern Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1998.
4. S. Kleiman and S. Devang. *Programming with Threads.* Prentice Hall, 1996.
5. S. Lankes. *Parallelisierung einer Komponente eines Flugplanungs-Codes auf einem speichergekoppelten PC-Cluster.* Diploma Thesis (in German), Chair for Operating Systems, RWTH Aachen, Germany, April 1998.
6. T.A. Ngo and L. Snyder. *Data Locality on Shared Memory Computers under Two Programming Models.* Technical Report 93-06-08, Univ. of Washington, Dept. of CS, and IBM Research Report RC19082, 1993.
7. S.M. Paas, M. Dormanns, T. Bemmerl, K. Scholtyssik, and S. Lankes. Computing on a Cluster of PCs: Project Overview and Early Experiences. *Proc. 1. Workshop Cluster Computing,* Technical Report CSR-97-05, TU Chemnitz, Dept. of Computer Science, Nov. 1997.
8. O. Sinnen. *Loop-Scheduling und Splitting-Verfahren auf NUMA Multiprozessoren.* Diploma Thesis (in German), Chair for Operating Systems, RWTH Aachen, Germany, 1997.
9. S. Tholen. *Parallelisierung raumakustischer Simulationsalgorithmen für SCI-Cluster.* Diploma Thesis (in German), Chair for Operating Systems, RWTH Aachen, Germany, 1998.

17. True Shared Memory Programming on SCI-based Clusters

Martin Schulz

Lehrstuhl für Rechnertechnik und Rechnerorganisation, LRR–TUM
Institut für Informatik, Technische Universität München
email: schulzm@in.tum.de
http://wwwbode.in.tum.de/Par/arch/smile/

17.1 Introduction

Due to their excellent price–performance ratio, clusters built of commodity off–the–shelf PCs and connected with low–latency network fabrics are becoming increasingly commonplace and are even starting to replace massively parallel systems. According to their loosely coupled architecture, they are traditionally programmed using the message passing paradigm. This trend was further supported by the wide availability of high–level message passing libraries like PVM [3] and MPI [20] and intensive research in low–latency, user–level communication architectures [8, 26, 34].

Besides the message passing paradigm, which relies on explicit data distribution and communication, a second one exists, the shared memory paradigm, which offers a global virtual address space for sharing data between processes or threads. It is preferably utilized in tightly coupled machines like SMPs that offer special hardware support for shared memory. It is generally regarded as the easier programming model, especially for programmers who are not used to parallel programming, but it comes at the price of higher implementation complexity either in the form of special hardware support or in the form of complex software layers. This prohibits the widespread use of shared memory programming models on cluster architectures as the necessary hardware support is generally missing.

Both programming paradigms have a large application base. Therefore, a comprehensive software infrastructure constructed for a specific platform has to support both paradigms efficiently. Only this ensures the broadest possible applicability of an architecture to already existing programs and gives the freedom of paradigm choice for implementing new parallel codes. The existence of such a software base is therefore a main component for the success of any new architecture.

However, the shared memory paradigm is mostly neglected on cluster architectures. Even SCI clusters, despite SCI's hardware distributed shared memory (HW-DSM) capabilities, are traditionally programmed using the message passing paradigm. This deficiency is caused by SCI's inability to provide a global virtual memory abstraction which is the prerequisite for any

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 291-311, 1999
© Springer-Verlag Berlin Heidelberg 1999

true shared memory model. This gap is bridged in our work by merging SCI's remote memory access capabilities with techniques from traditional software DSM systems resulting in a global transparent virtual memory which we call SCI Virtual Memory or SCI-VM. This system provides a flexible base for the development and implementation of true shared memory programming models on top of SCI [14].

This chapter describes the endeavor of implementing the SCI-VM as a software layer on Windows NT and standard Dolphin PCI-SCI adapter cards. Based on this software layer, the global address space can be combined with different execution models and synchronization constructs to form a variety of shared memory programming models. This is demonstrated here using two selected programming models, the concepts of which are described in depth as well. Models like these enable clusters to execute true shared memory applications without major porting efforts which closes the gap between tightly and loosely coupled systems.

The remainder of this chapter is organized as follows. Section 17.2 presents the basic design and the core mechanisms of the SCI-VM, followed by a discussion of the challenges involved in this design in Section 17.3. Section 17.4 shows how the SCI-VM can be utilized to implement shared memory programming models. Section 17.5 then presents such a programming model based on the SPMD principle, with performance results of SPMD programs given in Section 17.6. In Section 17.7 an overview of a second SCI-VM programming model, a fully transparent thread library for clusters, is given which constitutes an SMP-like environment on top of SCI clusters. The chapter is then concluded by a brief overview of related work in Section 17.8 and by some final remarks in Section 17.9.

17.2 Designing a Global Virtual Memory

The main prerequisite for shared memory programming on clusters is a fully transparent, global memory providing applications with the abstraction of a cluster–wide global process with a single virtual address space. This section describes a software layer that provides such a cluster-wide memory abstraction, the SCI Virtual Memory or SCI-VM (see also [19, 28]). It forms the basis for any kind of shared memory programming model on top of SCI-based PC clusters.

17.2.1 Building Block 1: SCI-based Hardware DSM

The SCI-VM should be directly based on the HW-DSM provided by SCI rather than deploying a traditional software DSM system with all its problems like false sharing and complex differential page update protocols [18]. Only this exploitation of HW-DSM enables the SCI-VM to benefit from the

special features and the full performance of the interconnection technology. Additionally, the implementation of synchronization primitives should directly utilize atomic transactions provided by SCI to ensure greatest possible efficiency.

17.2.2 Building Block 2: Software DSM Systems

Unfortunately, SCI alone cannot provide a global virtual memory abstraction as required by shared memory programming models. Both its hardware and software components only target the utilization of large, contiguous, and permanently pinned memory segments. In order to overcome these limitations and to reach a fully transparent implementation of a global virtual address space, the SCI remote memory capabilities have to be augmented by concepts and mechanisms well known from traditional software DSM systems, like data distribution at page granularity, on–demand access to remote pages, and a relaxed consistency model [24].

17.2.3 Combining Both Building Blocks to the SCI-VM

Together, the building blocks described above allow the formation of the transparent virtual address space which we call SCI-VM. The memory resources are distributed at the granularity of pages and these distributed pages are then combined into a single global virtual address space. In contrast to purely software based systems though, no page has to be migrated or replicated. All remote pages are simply mapped using SCI's HW-DSM mechanisms and then accessed directly. Due to the large amount of necessary mappings (in the worst case one for each remote page), these mappings are handled on demand with a similar concept as realized in the paging mechanisms of modern operating systems (OSs). The SCI-VM therefore represents a cluster–aware extension of an OS's virtual memory management.

This concept is further illustrated in Figure 17.1 for a two–node system. In order to establish a global virtual memory abstraction, a global process abstraction also has to be built with team processes as representatives for the global process on each node. These processes are running on top of the global address space which is created by mapping the appropriate pages from either the local physical memory in the traditional way or from remote memory using SCI HW-DSM.

The mapping of the individual pages is done in a two–step process. First, the requested page has to be located in the SCI physical address space from where it can be mapped into the PCI address space using the address translation tables (ATT) of the SCI adapter cards. From there, the page can be mapped with the help of the processor's page tables into the virtual address space of the local team process. The different mapping granularity in these two steps poses problems; while the latter mapping can be done at page granularity, the SCI mappings can only be done on the basis of segments in the

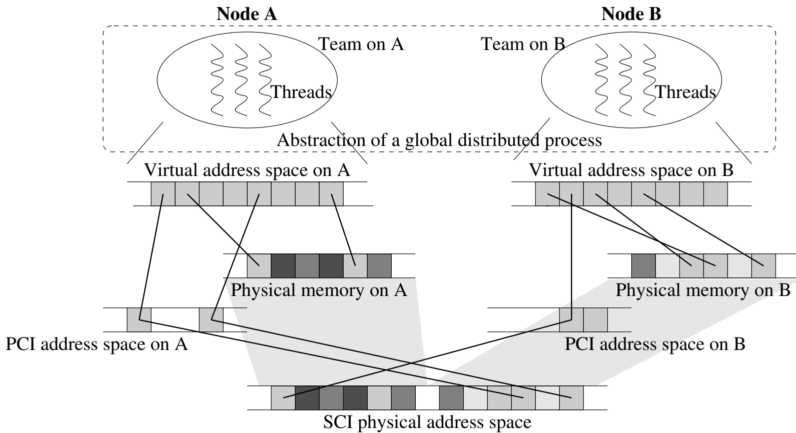


Fig. 17.1. Principal design of the SCI Virtual Memory

size of 512 kByte or 128 pages. This limitation is due to the current implementation of the PCI-SCI adapter cards [6]. To overcome this difference, the SCI-VM layer has to manage the mappings of several pages from one single SCI segment. The mappings of the SCI segments themselves will be managed with an on–demand, dynamic scheme very similar to paging mechanisms in OSs.

17.2.4 Locality Issues and Caching

The main potential problem in the SCI-VM is the issue of data locality. Although SCI achieves extremely low latencies, even when compared to other SAN networking technologies, remote memory reads still take one to two orders of magnitude longer than accesses to the local memory. This problem is further increased by the fact that SCI remote memory can by default not be cached as the PCI bus is not capable of snooping the memory bus thus prohibiting the implementation of the optional cache coherence protocol defined in the SCI standard [11].

Each access to the SCI remote memory would therefore suffer from the full latency penalty of the SCI network, which is roughly 5μs round-trip, causing a major performance problem for applications which are transparently distributed on top of the SCI-VM. One of the possible solutions is to incoherently enable caching of remote SCI memory and manage the incoherences by applying a relaxed memory consistency model. This would allow to take advantage of the rich memory hierarchy of modern computer systems reducing the impact of the poor performance of remote memory operations. As can be seen later in Section 17.6, some applications only suffer from overheads in the range of a few percent in this model.

17.3 SCI-VM Implementation Challenges

The design discussed above presents several interesting implementation challenges. The most severe is related to the integration of the concepts presented above into the underlying OS, in this case Windows NT and its virtual memory manager (VMM).

17.3.1 Mapping of Individual Page Frames

In order to implement the SCI-VM system, it has to be possible to map arbitrary physical memory locations into the virtual memory address space of the team processes. For this purpose, Windows NT offers the concept of Section Objects [5], which allow to map sections of physical memory in a way similar to memory–mapped files. These kind of mappings, however, exhibit two significant problems: (1) physical regions have to begin at 64 kByte boundaries, preventing the programmer from specifying single arbitrary pages, and (2) memory can only be mapped into non-committed memory areas of the virtual address space. This means that the SCI virtual memory concepts cannot be applied to already existing memory segments, preventing the distribution of static data that has been allocated and initialized at the load time of the process.

As long as Windows NT, like most other mainstream OSs, does not offer the required functions described above, the only way to overcome these shortcomings is to perform mappings at page granularity directly at processor level, bypassing the OS. This can easily be done with the help of a kernel mode driver. After getting access to the page directory via the Pentium II page directory base register (PDBR) [12], the address of the corresponding page table for a specific virtual page can be computed and then manipulated. When doing this, the flags representing the page's state and properties have to be adjusted appropriately. Specifically, the cache attributes have to be set.

This approach, however, has some severe consequences for the OS. As it is completely bypassed and therefore unaware of the manipulations, its stability and robustness may be impaired. However, based on several experiments using first implementations of the SCI virtual memory concept, Windows NT has proven to exhibit an acceptable stability. The critical point during the execution of a program using direct page mappings is its termination. While freeing a process' memory resources, Windows NT attempts to clean up its own page table entries. When it comes across unexpected entries introduced by the SCI-VM that are not consistent with its internal data structures, it generates a kernel panic and halts the processor. Especially in connection with memory committed by dynamic link libraries, this problem can be observed. It is therefore a vital necessity to erase all own entries from the page tables before terminating the program. This guarantees a clean program termination and ensures the stability of the OS. These results, however, are only

preliminary as detailed information on the internals of the Windows NT memory manager is presently not available to us. Due to this, the implications on the OS have not yet been analyzed in detail.

17.3.2 Dynamically Paged Memory

Closely related to the issue above is the utilization of dynamically paged memory. Only this allows a true extension of virtual memory to a cluster–wide virtual memory. This, however, causes an additional problem: as SCI only provides hardware DSM on the basis of each node's physical memory, only virtual memory pages which are pinned into a physical page frame can safely be accessed via SCI. To still allow the utilization of pageable memory, a dynamic locking scheme has to be applied. A fixed, configuration–dependent number of pages is kept pinned in physical memory and only these pages are allowed to be imported by other nodes. Whenever an additional page not included in the set of locked pages is requested, one pinned page is selected to be unpinned. At this moment, any mapping existing onto this page from any remote node is invalidated and marked as not present on these nodes, thereby making it safe to unpin the page on the local node. This then makes the page again eligible to be paged out of memory by the OS.

The notification of all nodes of the unpinning of a page is a global operation and therefore costly. However, the performance impact is likely to be comparable to that of local paging to secondary storage. Due to this, programs using the SCI-VM should, like standard sequential programs, stay within the limitations of the available physical memory. The possibility to utilize paged memory, however, allows the programmer to compensate for momentary peaks in memory utilization and allows program testing on smaller cluster configurations. It therefore adds to the convenience provided by the SCI-VM and aids in achieving the goal of full transparency and SMP–like behavior for clusters. The techniques used for this endeavor are very similar to those used to implement distributed, memory mapped file systems as described in Chapter 18.

17.3.3 Enabling Caching Using Relaxed Consistency

An additional problem is imposed by the fact that, although the SCI standard [11] defines a complex and efficient cache coherency protocol, it is not possible to cache remotely mapped memory in standard PC-based SCI clusters. This is caused by the inability of PCI-based SCI adapter cards to perform bus snooping on the system bus. However, caching is necessary to overcome the problem of the large latencies involved when reading transparently from remote memory. The only solution here is to apply a relaxed consistency model [1] that allows to enable caching while coping with the possible cache inconsistencies.

In order to allow for the greatest possible flexibility, the SCI-VM layer does not enforce a specific cache consistency protocol, but merely provides mechanisms that allow applications or higher level programming models to construct application–specific consistency protocols. For this purpose, the SCI-VM offers the functionality to flush the current memory state of the local node to the SCI network and to synchronize the memory of individual nodes with the global state by invalidating all local buffers including the caches. This functionality is implemented through mechanisms that allow to control both the SCI adapter card with its internal buffers and the CPU's memory model by providing routines to flush the local caches and write buffers.

17.4 Framework for SCI-VM-based Programming Models

The SCI-VM, as it has been presented above, does not form a complete programming model that is suitable for the development of parallel programs. It only provides a global view onto the distributed memory resources of an SCI cluster combining them into one single virtual address space on all nodes. This memory abstraction can then be utilized to build a variety of shared memory programming models on top of it generating a full framework for shared memory programming models on top of SCI–based clusters. This section discusses the mechanisms and functionality provided by the SCI-VM to aid in such developments together with a brief overview of the current implementation status. Additionally, two examples for SCI-VM based programming models are introduced below in Sections 17.5 and 17.7, proving the functionality and flexibility of the SCI-VM concept.

17.4.1 SCI-VM Interface

The interface exported by the SCI-VM to support the development of higher level programming models can be split into two components: an interface for managing the distributed shared memory and an interface to control the SCI-VM environment. The latter one is indispensable, as the SCI-VM creates not only a global virtual address space, but also a global process abstraction that holds this global virtual memory. Therefore, the SCI-VM is directly connected with all issues regarding the configuration of the cluster and the management of this process abstraction. This part of the interface includes routines to query the number of participating compute nodes, the local node number, to control process attributes, and to ensure a global process termination across all participating OS instances. In addition, this interface also includes a simple messaging mechanism that allows to invoke call-back routines on remote nodes as this is useful for the implementation of most programming models.

The interface giving access to the actual shared memory functions of the SCI-VM consists only of a few routines. The core is a routine for allocating memory in the global virtual address space. In its simplest form, it behaves like a *malloc* call, transparently allocating memory that is striped across all nodes at the finest possible granularity. An additional, optional parameter provides the capability of influencing the memory distribution of the newly allocated memory. This enables the user of the SCI-VM to optimize the data locality of the application if required or beneficial, while still allowing complete transparency whenever no data locality information is available.

In addition to this core routine for memory allocation, a few support routines are provided. They deal with giving access to a statically allocated configuration memory and with providing full transparency also for statically allocated variables. Those routines are normally only necessary at program or library startup time for initialization purposes.

17.4.2 Tradeoff Between Transparency and Performance

As already mentioned, this interface provided by the SCI-VM forms the basis for the development and implementation of any shared memory programming model on top of an SCI cluster. This can range from simple models with explicit resource control to fully transparent models providing an SMP-like environment for SCI clusters and can include systems with special compiler support. In any case, each specific programming model represents a tradeoff between transparency, which provides convenience for the programmer, and direct and explicit resource control, which allows for optimizations to achieve best possible performance. This tradeoff has to be carefully chosen to fit the needs of the programmers for which the model is intended.

17.4.3 Current Status of the Framework

The implementation of the SCI-VM and its shared memory programming model framework is currently in an early state and not yet complete. The final architecture is shown in Figure 17.2. The main part of the SCI-VM will be implemented as a Windows NT kernel mode driver based on top of two further drivers, the SCI device driver and a VMM driver to augment the Windows NT virtual memory management as described above. The kernel mode functionality is provided to the user through a user–mode library implementing the interface described above.

At this time, the SCI-VM core is still realized in user level on top of a preliminary kernel/user–mode interface. In kernel mode, only the two lower–level drivers exist and are directly used from user level to construct the global virtual memory abstraction. The user–mode part is currently, although conceptually separated, still implemented together with the existing programming model, the SPMD model discussed in the next section. This affects

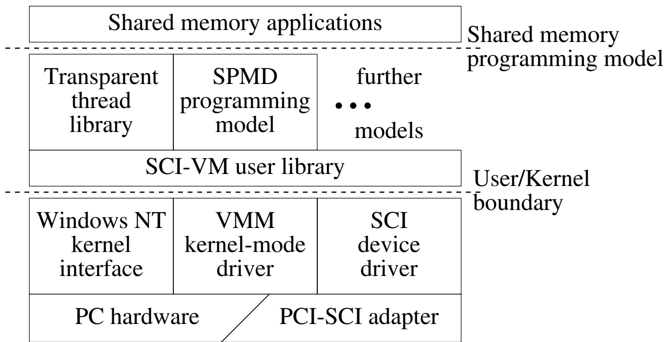


Fig. 17.2. System architecture including the SCI-VM and supported programming models

mostly the memory allocation routine which is implemented in a specialized synchronized version for this programming model. Its detailed description is also included in the next section.

This preliminary implementation of the SCI-VM architecture was chosen to ease the work on a first working SCI-VM that can be used for evaluation purposes and for debugging the VMM extension in kernel mode. Future versions will strictly separate the SCI-VM from the programming model as shown in the design presented above. Only this will provide the intended flexibility of the SCI-VM. This architectural refinement, however, will not have any impact on the performance compared to the preliminary version; the performance results presented below are therefore also valid for a full featured SCI-VM.

17.5 SPMD Programming Model on Top of SCI-VM

The first programming model implemented using the SCI-VM concepts is a model designed according to the *Single Program Multiple Data* (SPMD) principle. This model was chosen due its simple execution model which makes it ideal for experimenting and debugging, while still allowing the evaluation of the transparency of the SCI-VM.

17.5.1 The Execution Model

The SPMD programming model realizes a very simple concept of parallelism by allowing one thread of activity based on the same binary image of the executable per node. Due to this execution mode and due to the synchronous manner of allocating global resources, which will be discussed below, this programming model can be classified as a Single Program Multiple Data

(SPMD) model. Within the actual parallel execution, however, this synchronism is not fully enforced; it is possible to implement independent threads of control on each node as long as the restrictions regarding resource allocation are observed.

17.5.2 Allocating Shared Memory

The SPMD programming model offers a central routine to allocate shared memory. It reserves a virtual address segment that is valid on all nodes within the cluster and performs the appropriate mappings of interleaved local and remote memory into this address range. The memory resources are thereby distributed in a round robin fashion at page granularity, i.e., at a granularity of 4 kByte (on x86 architectures). This results in an even distribution of the memory resources across the cluster while trying to avoid locality hot spots.

This allocation routine has to be called by all nodes within the cluster simultaneously with the same parameters. It is therefore the main constraint that causes the SPMD properties of the programming model. This is currently necessary to request memory resources from all nodes during the allocation process. In the future, this constraint will be eliminated by a more complex SCI-VM implementation which allows to interrupt remote nodes to request remote memory.

The allocation itself is a three step process. In the first step, the contribution of the local node to the new global segment is determined and an appropriate amount of local physical memory is allocated. In the second step, each node enters the physical locations of all local pages together with its node ID into a global list of pages. After the completion of this operation on all nodes, a virtual address segment is allocated at the same location on each node and all pages in the global list are mapped into this newly created address segment. Local pages are mapped directly from physical memory, whereas remote pages are first mapped into the PCI address range and from there into the virtual address space. Upon completion of the page mappings on all nodes, the allocation process is completed and the new virtual address is returned to the caller.

17.5.3 Synchronization

Shared memory programming models need mechanisms to coordinate accesses to shared data and to synchronize the execution of the individual threads. For this purpose, the SPMD SCI-VM programming model provides the following three mechanisms: global locks without any association to data structures, cluster–wide barriers, and atomic counters that can be used for the implementation of further, specific synchronization mechanisms.

All of these mechanisms are implemented based on atomic fetch–and–increment transactions provided directly in hardware by the SCI adapter

cards in the form of designated SCI remote memory segments. Read accesses to these segments automatically trigger an atomic fetch–and–increment. This transaction atomically increments the value at the memory location specified in the read transaction and returns the original value of the memory location as the result of the read operation.

While the atomic counters can be directly implemented using the atomic SCI transactions, locks and barriers need additional concepts. Locks are implemented using two separate counters, a ticket counter and an access counter. Each thread trying to acquire a lock requests a ticket by incrementing the ticket counter. The lock is granted to the thread if and only if the value of the ticket counter equals the value of the access counter. On release of the lock, the thread increments the access counter and thereby grants the next waiting thread access to the lock.

The implementation of barriers utilizes one global atomic counter that is incremented by each thread on entry into the barrier. When the counter reaches a value that is divisible by the number of threads in the cluster, all threads have reached a barrier and are now allowed to continue. To distinguish multiple generations of one barrier, an additional local, non-shared counter is maintained. This counter keeps track of the number of times the barrier has been executed.

Currently, all waiting operations required by locks and barriers are implemented using simple spin waits. This is not a major problem as there is currently only one thread per node. In the future, however, this will be replaced using blocked and spin-blocked waits with a tight integration into the thread scheduling policies to allow more flexibility.

17.5.4 Consistency Model

The SPMD programming model utilizes a relaxed consistency model to enable the caching of remote memory. Most of the consistency enforcing mechanisms are implicitly embedded into the shared memory synchronization operations to ease the use of the programming model as this hides the relaxed consistency model from the direct view of programmer and instead offers the familiar shared memory synchronization primitives.

Barrier operations implicitly perform a full synchronization of all memory within the cluster. This is achieved by performing a flush of the current memory state to the network together with a complete invalidation of caches and local buffers on all nodes.

The consistency enforcing mechanisms embedded into locks provide a semantics similar to release consistency [1]; after acquiring a lock, the local memory state is invalidated, and before the unlock or release operation, local buffers are flushed to the network to force the propagation of the local memory state to remote nodes. This semantics guarantees that accesses to shared data structures, which are guarded by locks to implement mutual exclusion, are always performed correctly in a consistent manner.

In addition to this embedded consistency, the SPMD programming model also provides a routine for explicit memory synchronization. This routine performs a full flush and invalidation of the local memory on the node it has been called. However, this routine should only be necessary in special cases when working with unguarded data structures as all other cases are already covered by the embedded consistency enforcing mechanisms.

17.6 Experiments and Results

In order to evaluate the concepts presented above, we implemented a few small numerical kernels and ported an existing shared memory–based volume–rendering application using the described SPMD programming model. All codes utilized the shared memory in a fully transparent fashion without any locality optimizations. Also, the relaxed consistency model was applied only through the synchronization mechanisms described above without requiring any code modifications.

17.6.1 Experimental Setup

All of the following experiments were conducted on an SCI cluster in a two node configuration. This reduced version of the cluster was chosen due to current limitations in the implementation. Future version will overcome these limitations and are expected to scale to significantly larger cluster configurations. Each compute node in the cluster is a Pentium II (233 MHz) based PC with a 512 kByte L2 cache. The motherboards of these systems are based on the Intel 440FX chip-set. We deployed the PCI–SCI adapter cards, Revision D from Dolphin ICS [6]. These cards are equipped with the Link Controller LC-2 which allows to operate the SCI network at a raw bandwidth of up to 400 MByte/s per link.

The OS for all experiments was a standard Windows NT 4.0 (Build 1381, SP 3). The driver software for the SCI adapter cards was also supplied by Dolphin ICS through the SCI Demokit in version 2.06.

17.6.2 Results for the Numerical Kernels

The first experiments were conducted using three small numerical kernels: linear sum, standard dense matrix multiplication, and one iteration of the successive over-relaxation (SOR) method. All of these kernels make use of one contiguous virtual memory segment in which the program's data structures are mapped.

For each of these codes, three different experiments were performed: a parallel version on top of the global memory provided by the SCI-VM, a sequential version on top of local memory to get a baseline comparison for

the speed–up values, and a sequential version on top of global memory to get information about the overhead incurred by using the global memory abstraction. Based on the results from these experiments, two key values were computed: *speed–up* as the ratio of sequential execution time on local memory to parallel execution time and *overhead* obtained from the ratio of sequential execution time on global memory to that on local memory. The results for two different sizes of the virtual segment are presented in Table 17.1.

	Memory size	Local seq.	Speed–up	Overhead
Linear sum	256 kByte	4128 μs	0.92	63.01 %
	1024 kByte	16549 μs	1.06	53.56 %
Matrix multiplication	256 kByte	4912 ms	1.97	0.48 %
	1024 kByte	42731 ms	1.89	5.2 %
SOR iteration	256 kByte	296 ms	1.53	11.1 %
	1024 kByte	1225 ms	1.58	5.3 %

Table 17.1. Performance results for the numerical kernels

The linear sum performs poorly; applied to small problem sizes, no speed–up, but rather a slight slow–down can be observed. Only when applied to larger problem sizes, a small speed–up is possible. This can be explained by the fact that this algorithm traverses the global memory range only once and therefore does not utilize any temporal locality through the caches. The only applicable mechanisms to increase the performance are the implicit prefetching of cache lines and the prefetching facilities that are implemented within the SCI hardware. The result is a rather high overhead and consequently a low speed-up. In addition, the linear sum is an extremely short benchmark which causes the parallel version of the algorithm to infer significant overhead. This can only be improved by applying the algorithm to larger data sizes.

The matrix multiplication algorithm, however, behaves almost optimally by achieving a nearly perfect speed–up. This can be explained by the high exploitation of both spatial and temporal cache locality which is documented in the extremely low overhead numbers. When applying the scheme to larger data sizes, however, this overhead increases as the working set no longer fits into the L2 cache. Due to this, an increased number of cache misses has to be satisfied through the SCI network which causes additional overhead. This also leads to a slight decrease of the overall speed–up.

The third numerical kernel, the SOR iteration, shows a very good performance as well. This is again the result of an efficient cache utilization with respect to both temporal and spatial locality. In contrast to the matrix multiplication, however, it is not necessary to keep the whole virtual address segment within the cache, but rather only a small environment around the current position within the matrix. Due to this, the algorithm does not suffer

in performance when applied to data sizes larger than the L2 cache. It even benefits from the reduced overhead due to the larger data size, resulting in a higher speed–up.

17.6.3 Results for the Volume Rendering Code

While the experiments using the small numerical kernels provide detailed information about the raw performance and the problems of the SCI-VM and the SPMD model implementation, it is also necessary to perform experiments using complex applications to evaluate the full impact of the SCI-VM. For this purpose we utilized a volume rendering code from the SPLASH-II suite [35]. It was ported to the SPMD programming model by providing an adapted version of the ANL macros[1]. The actual volume rendering code itself was not modified.

To evaluate this application, the same experiments were conducted as described above for the numerical kernels. The results are summarized in Table 17.2. The values correspond to rendering times of single images. Pre- and post-processing of the data was not included in the measurements. The data set used for all experiments is the standard test case provided together with the SPLASH distribution and has a raw size of roughly 7 MByte. The work sharing granularity is set to blocks of 50x50 pixels to achieve the optimal tradeoff between management overhead and load balancing.

Sequential execution (local memory)	3522 ms
Sequential execution (global memory)	4136 ms
Parallel execution (global memory)	2381 ms
Speed–up	1.48
Overhead	17.43 %

Table 17.2. Performance results for the volume rendering application

This complex application which handles large data sets also exhibits a good performance on top of the transparent virtual address space. It is possible to achieve a speed–up of about 1.5 due to a low overhead caused by utilizing the transparent memory, of only about 18 %. Most of the overhead prohibiting a larger speed–up is caused by the management and locking overhead of a central work queue which is a common bottleneck in any shared memory environment. This experiment shows that the concepts presented in this paper can be directly and efficiently applied to a large existing shared memory

[1] The ANL macro package provides the basic shared memory constructs to the SPLASH applications in a platform–independent way. A dummy version for single processor systems and a sample version for SGI systems are provided with the SPLASH distribution [31]. Unfortunately, there is no further documentation available.

code without a major porting effort. Future experiments will extend this to further applications from different domains and will evaluate the SCI-VM in a variety of scenarios.

17.7 Using the SCI-VM for Transparent Multithreading

Besides the first implemented programming model on top of the SCI-VM, the SPMD model described in Section 17.5, several others with different capabilities and target domains are possible. One of the most interesting option is the implementation of a fully transparent, distributed thread library on top of the SCI-VM (see also [27]). The advantages and problems of such a model are described in this section. The concept presented here is currently being implemented within the ESPRIT project SISCI [7, 29] using the thread interface defined in the POSIX standard [33] and the thread interface defined by Microsoft's Win32 API [21]. The result will be a completely transparent thread library, the SISCI–Pthreads/Win32Threads, which establishes a true SMP environment on top of an SCI cluster in a fully transparent fashion. This will ease the port of existing multithreaded applications onto clusters and therefore open this architecture to a whole group of new applications.

17.7.1 Transparent Thread Distribution

The basis for the implementation of these thread libraries is the global virtual memory provided by the SCI-VM. It creates a consistent view onto the complete virtual memory from any node within the system and guarantees that any data is available on any node using the same address.

On top of this global memory abstraction, the global thread abstraction can be built. This thread abstraction has to be capable of creating and terminating remote threads and of modifying attributes of any thread in the system from any node. These functions will be implemented using a forwarding mechanism which forwards thread function calls to the team process hosting the thread in a remote method invocation–style manner. The only exception to this is the thread creation routine. Here a target node for the new thread has to be selected prior to the forwarding of the request. Currently this is done using a round–robin scheme to create an even thread distribution suitable for our homogeneous cluster environment. This, however, can easily be adapted to a more flexible and adaptive scheme which takes factors like load, interactive usage, and processing speed of the nodes into account.

A special problem in a distributed thread system is the program or process termination mechanism. Both POSIX and Win32 define that a process is terminated when its last thread terminates. This semantics has to be extended to the distributed thread implementation. This can be achieved by maintaining a global active–thread counter. When this counter reaches zero,

all other nodes are notified and the team processes terminate themselves. In addition, team processes have to kept alive, even after their last local thread has terminated, as threads on other nodes may still be active and/or there may be future requests to spawn a new thread on the local node. This is implicitly achieved by maintaining an additional thread per team process that is not part of the application threads. This does not cause any extra overhead, as such a thread is already required to handle the communication with the other team processes.

17.7.2 Synchronization Mechanisms

The POSIX thread API, as defined in [33], and the Win32 API, as defined in [21], include a variety of concepts that can be utilized to synchronize the execution of concurrent threads. The three main mechanisms are mutexes (both APIs), condition variables (POSIX) and events (Win32). The first mechanism can be used to guarantee mutual exclusion for the execution of critical sections, while the other two allow to signal events to other waiting threads. Similar to the SPMD programming model, these synchronization mechanisms can also be implemented using the SCI atomic transactions. In contrast to the SPMD model with one thread per node, however, a simple notification mechanism purely based on polling is not advantageous in a fully multithreaded environment; as there can be multiple threads per processor, a polling thread can seriously hurt the performance of the other threads. The notification mechanism, therefore, needs to be enhanced to cope with this problem, i.e., trading off the responsiveness of the waiting thread and the overall performance of the system. Possible solutions for this problem can range from exponential back-off mechanisms to remote interrupts. Future research will show the best performing solution for this special environment.

17.7.3 Applying a Relaxed Consistency Model

Special care has to be taken to incorporate the relaxed memory consistency model provided by the SCI-VM. As discussed in Section 17.2 and as seen in the results of the first experiments in Section 17.6, this is a necessary prerequisite to enable applications to run with an acceptable performance. The utilization of such a relaxed consistency model, on the other hand, has the severe side effect of changing the memory semantics for the applications.

In order to maintain full transparency in such a system, coherency enforcing constructs have to be applied rigidly which automatically provide a coherent memory state whenever the application might require it.

This endeavor is eased by the fact that the POSIX standard [33] already includes a relaxed memory coherence model. By mapping this relaxed consistency model onto the model provided by the SCI-VM, a POSIX–compliant, transparent shared memory is achieved. Therefore, all applications which are

cleanly written according to the POSIX standard will run transparently on top of the SISCI–Pthreads.

The POSIX consistency model is based on so–called synchronization points. These points are routines within the POSIX interface which have to enforce a fully consistent memory before their return. A complete list of these routines can be found in [33]. It mainly includes routines that deal with thread synchronization like mutex lock and unlock routines. These routines, which are implemented within the SISCI–Pthreads library, implicitly enforce this synchronization through the primitives provided by the SCI-VM. Although this approach does not offer the optimal solution with regard to performance, it guarantees a fully transparent execution of multithreaded codes on top of SCI.

17.8 Related Work

Work on shared memory models for clusters of PCs is mostly done in the area of pure software DSM systems. A well–known representative of this group is the TreadMarks [2] implementation. Here, shared pages are replicated across the cluster and synchronized with the help of a complex multiple-writers protocol. To minimize the cost of maintaining the memory consistency, a relaxed consistency model is applied, the Lazy Release Consistency [1, 15]. However, unlike in the SCI-VM approach, where the sharing of the memory is transparently embedded into a global abstraction of a process, TreadMarks requires the programmer to explicitly allocate either local or global segments. The same also holds for any other currently existing software DSM package, although they vary with respect to their APIs, consistency models, and usage focus. Examples of those kind of systems that have been developed for the same platform as the SCI Virtual Memory – Windows NT – are Millipede [13], Brazos [30], and the SVMlib [25].

Besides these pure software DSM systems that just deal with providing a global memory abstraction on a cluster of workstations, there are also projects that provide an execution model in the form of a thread library on top of the constructed global memory. One example for this are the DSM-Threads [23], which are based on a POSIX 1003.1c conforming API. This thread package, based on the FSU-Pthreads [22], allows the distribution of threads across a cluster interconnected by conventional interconnection networks. However, it also does not provide complete transparency. It therefore forces the user to modify and partly rewrite the application's source code.

Only a few projects utilize techniques similar to the SCI Virtual Memory and try to deploy SCI's hardware DSM capabilities directly. In the SciOS project (Chapter 18 and [16]), a global memory abstraction is created using SCI's DSM capabilities to implement fast swapping to remote memory. Researchers at the University of California at Santa Barbara are also working on providing a transparent global address space with the help of SCI (Chapter

15 and [9, 10]). Their approach, however, does not target a pure API, but rather is a hybrid approach of shared memory and message passing in the context of a runtime system for Split-C.

This work is in principle also applicable to any other non-cache-coherent NUMA architecture. One widely known commercial representative of this type of machine is Cray/SGI's T3D/E [17]. Also on this machine efficient low–level message passing, e.g., in the form of Fast Messages [26], and shared memory programming in the form of a restricted *put* and *get* functionality is available. However, no global virtual memory system like the SCI-VM presented above is provided. Other well known work for NUMA systems in academia can be found at Princeton University in the Shrimp project [4] and at the University of Rochester in the Cashmere project [32].

17.9 Conclusions and Future Work

Clusters of commodity PCs have traditionally been exploited using applications built according to the message passing paradigm. Shared memory programming models, which are generally regarded as easier and closer to sequential programming, are only available through software DSM systems. Currently, however, these systems lack performance and/or transparency. With the help of the SCI Virtual Memory presented in this chapter, it is now possible to create a fully transparent global virtual address space across multiple nodes and OS instances. This is achieved by combining hardware DSM mechanisms for physical memory with page–based techniques from traditional software DSM systems to provide the participating processes on the cluster nodes with an identical view onto the distributed memory resources.

The global virtual address space provided by the SCI-VM can then be used as a basis for implementing various kinds of shared memory programming models. Each of these models represents a tradeoff between full transparency with greatest possible convenience for the programmer and full control for optimum performance. This tradeoff has to be carefully chosen according to the needs of the programmer and the target application domain.

Two examples of such programming models based on the SCI-VM have been presented here: an SPMD–style model and a transparent thread library based on the POSIX/Win32 thread standard. The SPMD model provides the programmer with an easy to use parallel programming model that allows the execution of parallel applications on top of a global virtual memory in a synchronous manner. First results with several small numerical kernels and one larger volume rendering application show good performance results and prove the feasibility and applicability of the presented concepts. The latter model, which is currently being implemented, will allow the fully transparent execution of SMP applications on top of clusters of PCs. It is intended to function as a bridge between tightly coupled systems like SMPs and loosely

coupled clusters allowing for easy porting of thread based codes onto SCI–based cluster architectures.

In summary, the SCI-VM is the key component to open the cluster architecture to the shared memory programming paradigm which has been traditionally the domain of tightly coupled parallel systems like SMPs. Together with appropriate programming models atop, it will ease the programmability of clusters and allow to utilize the large number of existing shared memory codes directly in cluster environments.

Acknowledgments

This work is supported by the European Commission in the Fourth Framework Programme under ESPRIT HPCN Project EP23174 (Standard Software Infrastructure for SCI based Parallel Systems – SISCI). The SCI device driver source code has been kindly provided by Dolphin ICS. The driver software to augment the Windows NT memory management has been implemented by Detlef Fliegl (LRR–TUM, TU München). Last but not least, I would like to thank my wife Laura for proofreading and her support in general.

References

1. S. Adve and K. Gharachorloo. *Shared Memory Consistency Models: A Tutorial.* Rice University ECE Technical Report 9512 and Western Research Laboratory Research Report 95/7, Department of Electrical and Computer Engineering, Rice University, and Western Research Laboratory, DEC, Sept. 1995.
2. C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, Feb. 1995.
3. A. Bengelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. *A User' s Guide to PVM Parallel Virtual Machine.* Oak Ridge National Laboratory, Oak Ridge, TN, July 1991.
4. M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. 21st Int'l. Symposium on Computer Architecture*, volume 22 of *CAN*, pages 142–153, ACM, Apr. 1994.
5. H. Custer. *Inside Windows NT.* Microsoft Press, 1992.
6. Dolphin Interconnect Solutions, AS. *PCI–SCI Cluster Adapter Specification*, Version 1.2, May 1996.
7. M. Eberl, H. Hellwagner, B. Herland, and M. Schulz. SISCI — Implementing a Standard Software Infrastructure on an SCI Cluster. In W. Rehm, editor, *Tagungsband zum 1. Workshop Cluster Computing*, number CSR-97-05 in Chemnitzer Informatik–Berichte, pages 49–61, Nov. 1997.
8. M. Eberl, H. Hellwagner, W. Karl, M. Leberecht, and J. Weidendorfer. Fast Communication Libraries on an SCI Cluster. In *Proc. SCI-Europe '98, a conference stream of EMMSEC '98*, pages 165–175, Sept. 1998.

9. M. Ibel, K. Schauser, C. Scheiman, and M. Weis. Implementing Active Messages and Split-C for SCI Clusters and Some Architectural Implications. In *Proc. 6th International Workshop on SCI-based Low-cost/High-performance Computing*, SCIzzL, Sept. 1996.
10. M. Ibel, K. Schauser, C. Scheiman, and M. Weis. High-Performance Cluster Computing Using SCI. In *Proc. Hot Interconnects V*, Aug. 1997.
11. IEEE Computer Society. *IEEE Std 1596–1992: IEEE Standard for Scalable Coherent Interface*. The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, August 1993.
12. Intel Corporation. *Intel Architecture Software Developer's Manual for the PentiumII*, volumes 1–3. Published on Intel's Developer Website, 1998.
13. A. Itzkovitz, A. Schuster, and L. Shalev. Millipede: a User-Level NT-Based Distributed Shared Memory System with Thread Migration and Dynamic Run-Time Optimization of Memory References. In *Proc. 1st USENIX Windows NT Workshop*, Aug. 1997.
14. W. Karl, M. Leberecht, and M. Schulz. Supporting Shared Memory and Message Passing on Clusters of PCs with a SMiLE. In A. Sivasubramaniam and M. Lauria, editors, *Proc. CANPC'99*, LNCS 1602. Springer Verlag, 1999.
15. P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Rice University, Jan. 1995.
16. P. Koch, E. Cecchet, and X. de Pina. Global Management of Coherent Shared Memory on an SCI Cluster. In *Proc. SCI-Europe '98, a conference stream of EMMSEC '98*, pages 51–57, Sept. 1998.
17. R. Koeninger, M. Furtney, and M. Walker. A Shared Memory MPP from Cray Research. *Digital Technical Journal*, 6(2), 1994. http://www.digital.com/info/DTJE01/DTJE01SC.TXT.
18. H. Lu, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proc. Supercomputing '95*, Dec. 1995.
19. M. Schulz and H. Hellwagner. Extending NT Virtual Memory by SCI–based Hardware DSM. In *Proc. 2nd USENIX Windows NT Symposium*, Aug. 1998.
20. Message Passing Interface Forum (MPIF). *MPI: A Message-Passing Interface Standard*. Technical Report, University of Tennessee, Knoxville, June 1995. http://www.mpi-forum.org.
21. Microsoft Corporation. *Microsoft Platform Software Development Kit*, chapter About Processes and Threads. Microsoft, 1997.
22. F. Müller. A Library Implementation of POSIX Threads under UNIX. In *Proc. USENIX*, pages 29–42, Jan. 1993.
23. F. Müller. Distributed Shared Memory Threads: DSM–Threads, Description of Work in Progress. In *Proc. Workshop on Run–Time Systems for Parallel Programming*, pages 31–40, Apr. 1997.
24. B. Nitzberg and V. LO. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, pages 52–59, Aug. 1991.
25. S. Paas, M. Dormanns, T. Bemmerl, K. Scholtyssik, and S. Lankes. Computing on a Cluster of PCs: Project Overview and Early Experiences. In W. Rehm, editor, *Tagungsband zum 1. Workshop Cluster Computing*, number CSR-97-05 in Chemnitzer Informatik–Berichte, pages 217–229, Nov. 1997.
26. S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2), 1997.
27. M. Schulz. SISCI-Pthreads, SMP–like programming on an SCI–cluster. In *Proc. HPCN Europe '98*, Apr. 1998.

28. M. Schulz and H. Hellwagner. Global Virtual Memory based on SCI-DSM. In *Proc. SCI-Europe '98, a conference stream of EMMSEC '98*, pages 59–67, Sept. 1998.
29. SISCI Consortium. Standard Software Infrastructure for SCI-based Parallel Systems (SISCI). http://www.parallab.uib.no/projects/sisci/, Aug. 1997.
30. E. Speight and J. Bennett. Brazos: A Third Generation DSM System. In *Proc. 1st USENIX Windows NT Workshop*, Aug. 1997.
31. SPLASH Research Group at Stanford. SPLASH distribution. ftp://www-flash.stanford.edu/pub/splash2/, Nov. 1996.
32. R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. SOSP'97*, Oct. 1997.
33. Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating Systems Interface (POSIX) — Part 1: System Application Interface (API).* ANSI/IEEE Std. 1003.1. IEEE, 1995 edition, 1996.
34. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19th Int'l. Symposium on Computer Architecture*, May 1992.
35. S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH–2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Int'l. Symposium on Computer Architecture*, pages 24–36, June 1995.

18. Implementing a File System Interface to SCI

P.T. Koch[1], J.S. Hansen[1], E. Cecchet[2], X. Rousset de Pina[2]

[1] Department of Computer Science, University of Copenhagen, Denmark
email: {koch, cyller}@diku.dk
http://www.diku.dk/distlab
[2] SIRAC Laboratory, INRIA Rhône-Alpes, France
email: {Emmanuel.Cecchet, Xavier.Rousset}@inrialpes.fr
http://www.inrialpes.fr/

18.1 Introduction

This chapter deals with the issues in implementing a file system interface to the shared memory in an SCI cluster. We describe the possibilities of sharing in file systems and how it can be implemented for SCI in UNIX systems and Windows NT. We present our prototype, *SciOS*, which implements a memory-based distributed file system. We find that the file system interface integrates SCI well with the operating system and provides sharing mechanisms, a symbolic name space, and possibilities for protection on multi-user SCI clusters.

18.1.1 Motivation

SCI can be implemented as a normal *device driver* (see Chapter 9). This requires a minimum amount of code that can be ported to a wide range of operating systems. The programming interface consists of device control commands, e.g., `ioctl` calls to allocate and map remote memory, which are defined by the driver. Libraries can implement more easy-to-use interfaces on top of the device driver.

An alternative way is to implement SCI as a *network driver*. Applications can then use standard message-passing interfaces and protocols to communicate over SCI (see Chapter 12). This allows existing distributed applications written for message-passing to benefit from the high performance of SCI.

Physical memory can be given a *disk driver* interface and implement a RAM disk. Access to physical memory is much faster than to a disk and is thus well-suited for temporary files that do not need to be persistent. But it is inflexible because it occupies a large, fixed amount of physical memory which can result in poor utilization of system resources. A system like *Memory File System* [14] is integrated with the virtual memory system of the kernel which makes it more flexible in its use of physical memory. The memory-based file system approach can also be taken for SCI clusters which can implement *distributed* file systems based on cluster-wide shared physical memory.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 313-331, 1999
© Springer-Verlag Berlin Heidelberg 1999

18.1.2 SCI-based File Systems

Using a *file system* interface to the physical memory in an SCI cluster has a number of advantages. The definitions of file system interfaces are traditionally stable and have developed incrementally and with backward compatibility. The programming interface and tools are well understood by users, at least for non-distributed file systems. A file system provides mechanisms for the sharing of data, a flexible naming scheme with symbolic names, and control of resource utilization. The protection mechanisms provided in file systems are important on large, multi-user SCI clusters.

The traditional use of memory-based file systems is for temporary files that do not need to be persistent and survive system crashes. Because the files are kept in memory only, such file systems easily outperform normal file systems. Even if normal file systems do extensive caching of a file's contents in main memory, some of the file attributes must be stored on stable storage, thereby slowing down file creation and deletion. An example of a use of temporary files is a compiler which saves intermediate files between compilation phases. For an SCI-based file system, the use of non-persistent files goes beyond such temporary files that are accessed sequentially by different processes.

Typically, SCI clusters are used for parallel applications which can use shared files for holding matrices and other data structures that are accessed concurrently by multiple nodes in the cluster. Sharing may occur at a very fine granularity and consistency guarantees must be given for the application to produce correct results. The size of the data sets used in scientific applications may be much bigger than the physical memory and swap space that is available in a single node. With SCI, a memory-based file can be held in physical memory of multiple nodes and shared through the SCI hardware. Having a file interface to SCI allows easy post- and pre-processing of shared data with general tools such as *awk*, *perl*, *Tcl*, and editors. A file system provides only a *low-level* interface. More elaborate environments for developing and executing distributed and parallel applications are still needed for SCI.

18.1.3 Outline

Section 18.2 introduces and discusses sharing in file systems and Section 18.3 describes how to add a new file system to the UNIX and Windows NT operating systems along with the main issues for implementing a file system interface to SCI. Section 18.4 gives an example of our prototype, SciOS, which uses a file system interface to implement a shared virtual memory on a cluster of workstations running Linux and using Dolphin's SCI cluster adapters. Section 18.5 presents related work, and Section 18.6 summarizes our findings and conclusions.

18.2 Sharing in File Systems

Especially multi-user and distributed file systems have to deal with issues of file sharing. We now briefly discuss the topic of sharing in file systems in UNIX variants and Windows NT.

When multiple processes share a file with explicit I/O calls such as `read` and `write`, each process has to read in the shared data into local buffers. Modifications can only be made to the local buffer and they have to be explicitly written back to the file before other processes can see the modifications. The resulting delay can complicate sharing and it may incur serious overheads from a large amount of system calls and copying between user and kernel space. Memory mapping of files solves these problems for non-distributed systems.

18.2.1 Memory-Mapped Files

Memory-mapping of a file allows a process to access the content of a file directly in memory through normal load and store instructions. The operating system assures that the file's content is available in physical memory and allows multiple processes to map the content through virtual-to-physical mappings. This means that modifications made by one process become immediately visible to the other processes without explicit system calls or copying. An example showing two processes mapping the same file in UNIX is shown in Figure 18.1. Windows NT has the same functionality.

Memory-mapped files can be used to directly access *remote* physical memory in an SCI cluster. Direct access to remote memory may not have the same semantics as the access to local physical memory. This is discussed in more detail in Section 18.2.3 and Section 18.3.6.

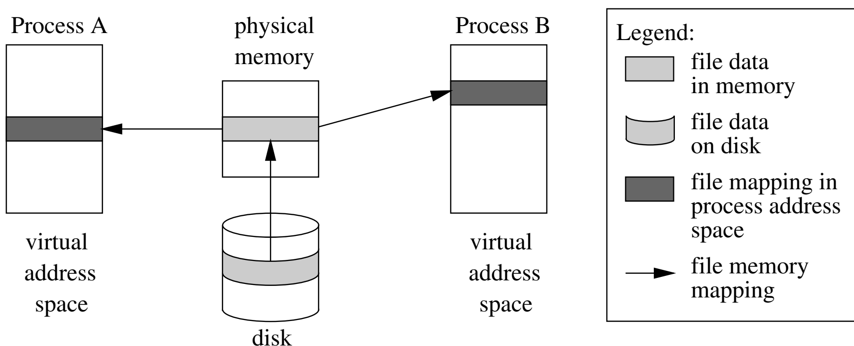


Fig. 18.1. Example with two processes mapping the same file in physical memory.

18.2.2 UNIX Example with a Memory-Mapped File

We illustrate memory-mapped files with an example in UNIX. A process creates and opens a file with the name `/scios/test/example` as shown in Figure 18.2. The file is created in such a way that the user has read and write permissions and the group has read permission only. The name space is shared and the `open` call initializes the file descriptor `fd`. The file is mapped into the virtual address space with the `mmap` call and is then initialized with the function `init_table`. The function is unaware that it is initializing a file and it can access the elements of `table` in a random order.

```
fd = open("/scios/test/example", O_RDWR|O_CREAT,
          S_IRUSR|S_IWUSR|S_IRGRP);
table = mmap(0, 2048 * sizeof(*table), PROT_READ|PROT_WRITE,
             MAP_SHARED, fd, 0);
init_table(table); /* initialize 2048 elements */
```

Fig. 18.2. A process creates, maps and initializes the file `/scios/test/example`.

Another process opens the file `/scios/test/example`, maps it and prints its contents (see Figure 18.3).

```
fd = open("/scios/test/example", O_RDONLY);
table = mmap(0, 2048 * sizeof(*table), PROT_READ, MAP_SHARED, fd, 0);
for (i = 0; i < 2048; i++)
    printf("%d\n", table[i]);
```

Fig. 18.3. A process opens, maps and reads an existing file `/scios/test/example`.

In the above example, the file is mapped at a virtual address determined by the operating system and this address may be different in all the processes that map the file. If the file needs to be mapped at a fixed virtual address, e.g., when the file contains virtual addresses, the process can give a virtual address to the `mmap` call and specify `MAP_FIXED`. A process may map the file multiple times and only parts of it specified by an offset and a length to the `map` call.

18.2.3 File Consistency

The normal consistency semantics for a UNIX file is that modifications are immediately visible to other processes accessing the file. It is known as *UNIX semantics*. This can easily be implemented on non-distributed systems, because the kernel can use the same kernel disk buffers for all processes and the

same physical memory for memory-mapped files. In distributed file systems, a number of performance optimizations are needed to avoid bottlenecks in the file servers or excessive communication. Client nodes may cache or replicate whole files or disk blocks to enhance locality and reliability. Clients may also delay sending modifications to the file server to reduce network traffic. Some distributed file systems often only provide *session semantics* where the modifications to a file are not visible on other nodes until the file is closed and subsequently opened on another node. There may also be inconsistency if the file is memory-mapped on multiple nodes, or if it is accessed simultaneously through a memory map and `read`/`write` calls.

An SCI-based file system must provide much stronger guarantees than session semantics. Parallel applications often expect memory consistency models known from shared-memory multiprocessors [4]. These range from sequential consistency (like UNIX semantics) to models where remote load and store instructions may be reordered until certain synchronization points have been reached in the application. The implementation of file consistency for SCI clusters is highly dependent on the capabilities of the SCI hardware, especially for remotely mapped memory. This is discussed further for the SciOS prototype in Section 18.4.

18.2.4 Synchronization

Synchronization is an area where the file system interface is not well suited for an SCI-based file system. The basic synchronization mechanism supported by non-distributed file systems is locking. In both UNIX variants and Windows NT, either a whole file or just a specified byte range may be locked. Usually both read and write locks are available. Both mandatory and advisory locking can be used. When mandatory locking is used, all operations that potentially can violate a lock must check for conflicts. One exception is the use of memory mapped files where locks are not enforced on memory operations due to the lack of efficient byte-range protection in the memory management system. For distributed file systems, file synchronization is complicated. File systems based on stateless servers, e.g., NFS, do not hold states for each open file, and can therefore not support distributed locking mechanisms. With NFS, this is solved by using a separate synchronization service. For parallel applications using an SCI-based file system, a wide range of synchronization primitives is needed. At least primitives like locks, condition variables, and barriers are expected by most parallel applications.

18.3 Issues for Implementing SCI-based File Systems

Modern operating systems allow several file system implementations to coexist. A file system can be added in a modular way, and the operating systems

provide a single interface, e.g., file operations, naming, and access control, to all of them. In UNIX systems, the most widespread file system framework is the virtual file system (VFS) which was originally designed by Sun [11] for the Network File System and it was later adopted by System V UNIX in SVR4 [6]. The VFS interface has evolved considerably and different UNIX systems provide extensions to the VFS interface, e.g., 4.4BSD and OSF/1. Non-UNIX operating systems also provide a file system framework, e.g., Windows NT which provides Installable File Systems and File System Drivers [15].

In the following, we examine the structure of the VFS interface in Linux 2.0 and UNIX System V Release 4 (SVR4) together with the Windows NT file system drivers. We refer to all the virtual file system frameworks as VFS and the UNIX implementations as vnode/vfs. We consider how an SCI-based file system can be implemented in these systems.

18.3.1 A Virtual File System

The primary role of a VFS is to link a number of different file systems together in a single name space, to provide a representation for objects residing on the file system, e.g., files and directories, and to convert the invocation of the system calls into operations on a particular file system. The separation between the system calls and the file system implementation is achieved by introducing a level of indirection. In vnode/vfs, this is called the *vnode layer*, and in Windows NT, it is handled by the I/O Manager.

The VFS maintains a file system independent representation of mounted file systems and their objects such as files and directories. In a vnode/vfs system, file system objects are represented by virtual nodes, *vnodes*, and in Windows NT, they are represented by special objects. Selected fields from a vnode data structure are listed in Table 18.1. The `v_op` field is a pointer to a list of operations which each file system can implement for directories and files, e.g., `lookup`, `open`, `read`, and `write` which are discussed in Section 18.3.4.

`v_count`	Reference count.
`v_vfsmountedhere`	Covering file system.
`v_op`	Pointer to vnode operations.
`v_vfsp`	Pointer to the file system of the vnode.
`v_type`	Vnode type.
`v_data`	Pointer to file system dependent data (an inode).

Table 18.1. Selected fields in a vnode data structure.

18.3.2 Files and Directories

UNIX systems and Windows NT use a hierarchical name space. A new file system can be made part of the global name space by *mounting* it to a directory—the mount point—in the existing name space. The directory of the original file system is then hidden and is changed to refer to the root directory of the new file system. Each file system has general information for describing a file system. In UNIX, it is maintained in a superblock data structure and Windows NT maintains a volume device object.

In Windows NT, it is the responsibility of the file system implementation to parse any pathname for a file, e.g., `/scios/test/example/`. In vnode/vfs, the vnode layer uses the VFS operation `lookup` to obtain a new vnode given a directory vnode and a symbolic name. Originally, the lookup operation only resolved one level of a file path name at a time. In distributed file systems this can result in considerable overhead, as each lookup is resolved across the network. In recent VFS implementations, such as 4.4BSD, multiple levels can be resolved at a time.

In vnode/vfs, the file system dependent state information of a file is often referred to as an index node or an *inode*. An inode contains status information and the data disk block numbers of the file. It is referenced from a vnode (see the field `v_data` in Table 18.1). In most file systems, a directory is a regular file containing a list of names and inode references (e.g., a pointer, a disk block number) for the files contained in the directory.

In Windows NT, the file structure and terminology is slightly different, but the basic representation is the same. The I/O Manager maintains a file object for each reference to an open file, and the file system implementation maintains a File Control Block for each open file which contains both file system independent and file system dependent information. This corresponds to a combination of vnode and inode information in vnode/vfs.

18.3.3 Example of Vnode/vfs Data Structures

In an SCI-based file system, the inodes and data pages should be placed in shared memory giving all nodes easy access to both file system structure and data. To keep the file system consistent the nodes must synchronize access to both inodes and data pages, e.g., if two processes add different entries to the same directory both the file size in the inode and the data pages must be updated atomically for each entry. Figure 18.4 shows the data structures resulting from simultaneously opening the SCI-based file `/scios/test/example` by two processes as shown in Figures 18.2 and 18.3. The two directory files, `/scios` and `/test`, and their vnodes are the result of the name resolution through one or more lookup operations. Inodes keep track of the location of the physical memory that makes up, e.g., the file `example`. For small files, a number of physical pages are reached directly from an inode. Larger files will require indirect inodes—like page tables in virtual memory systems.

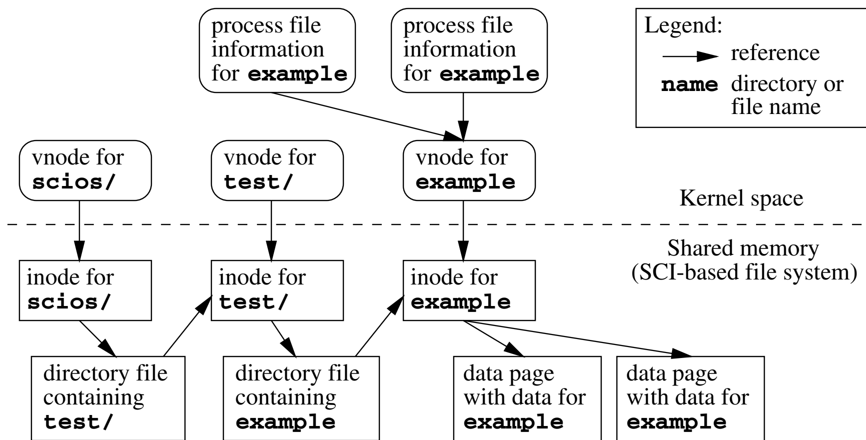


Fig. 18.4. Data structures for the open SCI-based file `/scios/test/example`.

18.3.4 Virtual File System Operations

For a new file system using a VFS interface, a number of operations need to be implemented. The operations are called by the VFS in response to system calls and other events in the operating system. There are operations on the file systems itself, e.g., `mount` and `unmount`, and operations on individual files, e.g., `open`, `read`, and `write`.

In vnode/vfs, the operations on directories and files are called directly through a vnode's `v_op` field (see Table 18.1). The most relevant operations are summarized in Table 18.2. Each kernel has its private file system independent data structures for the file, e.g., vnodes in vnode/vfs in the example in Figure 18.4.

When a process maps a file in memory, the `mmap` operation is called with the virtual address range of the new memory map. For a memory-mapped file, the `getpage` operation is called on a page fault and the `putpage` is called when flushing a dirty page. In addition the file system may use virtual memory operations (see Section 18.3.5) instead.

File access is verified through the `access` operation, and the file attributes are manipulated through the `getattr` and `setattr` operations. For an SCI-based file system conventional access control can be used, as SCI-based systems are usually tightly coupled and have the same set of users on each node.

In Windows NT, the I/O Manager communicates with the file system using I/O Request Packets. Such request packets include a command which is used to invoke a specific function in the file system. The Windows NT interface has fewer functions than vnode/vfs, but uses parameters more ex-

`open/create/close`	Open/create/close a file.
`link/remove`	Create/remove link to file.
`read/write`	Read/write data from/to a file.
`getattr/setattr`	Get/set attributes for a file.
`access`	Check access permissions for a file.
`ioctl`	Control operations.
`lookup`	Resolve name in directory to vnode.
`mkdir/rmdir`	Create/remove a directory.
`mmap`	Map a file into a process.
`getpage/putpage`	Read/write a page for a memory-mapped file.

Table 18.2. Selected vnode operations. The naming and semantics of the operations vary slightly between the different vnode/vfs implementations.

tensively, e.g., the `IRP_MJ_CREATE` command handles both `open`, `create`, and `mkdir` operations.

18.3.5 Interaction with the Virtual Memory System

Most file system implementations also need to access low-level kernel functions such as the virtual memory system, especially for memory-mapped files. The `getpage/putpage` in the VFS interface (see Section 18.3.4) does not give complete control of when physical memory is allocated and more advanced operations can be specified for virtual memory ranges.

Table 18.3 shows selected virtual memory operations for Linux. Similar operations exist for memory segments in SVR4. Page faults for pages with no virtual-to-physical mapping are handled by the `nopage` operation and swap operations handled by the `swapin` and `swapout` operations. The creation and manipulation of memory mapped files in Windows NT is not visible to the file system implementation. The virtual memory system maintains a Prototype Page Table (PPT) data structure for each mapped file. The PTT is created when a file is mapped and subsequent mappings of the same file reuse the existing PTT without involving the file system.

`unmap`	Unmap a range of pages from the address space.
`protect`	Change virtual memory protections on a range of pages.
`sync`	Write modified pages back to swap file or mapped file.
`nopage`	Handle page fault for a page with no mapping.
`wppage`	Handle write to a read protected page.
`swapout/swapin`	Place/fetch a page on secondary storage.

Table 18.3. Selected virtual memory operations for Linux.

For SCI-based file systems, additional virtual memory support is needed to manage mappings of remote memory through the I/O bus. One case that

needs to be handled is when a page is swapped out. Then all nodes that map the page remotely must invalidate their mappings so further accesses will result in page faults which can swap the page back in. This requires that the file system knows which processes on which nodes map the pages at which virtual addresses. In Windows NT, this information is not readily available.

18.3.6 Remote Memory Mappings and File Consistency

When multiple processes map a file on the same node, they share the same physical memory and they access the memory with the consistency guarantees of the local memory system. When a process maps a file on a remote node in an SCI-based file system, the load and store operations pass by the I/O bus and the SCI interconnect.

On current 32-bit I/O busses, the amount of address space available for remote memory mappings on a node is far from enough to map all the physical memory for the files in an SCI cluster. If multiple processes on the same node map the same remote page of a file, they can share the remote mapping in the SCI cluster adapter. But the file system has to deal with the case when all the available remote mappings have already been used. A solution is to manage the remote mappings as the operating system manages the TLB for virtual-to-physical memory mappings. To make room for a new remote mapping, an existing remote mapping can be invalidated and the virtual-to-physical mappings in processes that use that remote mapping have to be invalidated.

Most SCI clusters have weaker consistency models for remote memory accesses than for local accesses. For example, in a cluster with Dolphin's PCI-SCI adapters (see Chapter 3), writes to remote memory may be reordered. So when a process accesses a remote file, special *flush* operations are needed to make sure that writes have been performed in a remote memory. Also, if the local cache or prefetching is enabled to improve performance, old values must be flushed into the local cache or the PCI-SCI adapter when accessing memory that has been modified on another node. The solution is to specify a consistency model that is weak enough to deal with the consistency properties of the SCI cluster. One possibility is to specify *release consistency* [4] which allows writes to shared memory to be reordered, but requires that all accesses to shared memory are correctly synchronized. The file system must then provide synchronization primitives that deal with the correct flushing of caches and buffers in the SCI adapters to fulfill the consistency guarantees.

18.3.7 Synchronization

The amount of support for synchronization varies greatly between the different VFS implementations. In systems like SVR4 and Windows NT, the locking operations are visible to a VFS based file system, but in Linux, the locking mechanism is implemented directly in the VFS layer, and is therefore

not visible to the file system implementation. Although byte-range locks can be used by an SCI-based file system to provide locking of memory objects, this is typically not implemented for memory-mapped files. Furthermore, locks are not sufficient for parallel applications, which need more specialized primitives like condition variables and barriers. For these primitives, the `ioctl` control operation can be used to implement the required primitives which can be integrated with the shared memory consistency model.

18.4 The SciOS Prototype

The SciOS prototype implements a distributed shared virtual memory system on an SCI cluster. The prototype has a file system interface. SciOS currently runs on cluster of Intel PCs with Linux 2.0 and Dolphin's PCI-SCI Rev. D adapters. In this environment, the difference between local and remote memory accesses is high and SciOS provides mechanisms for automatic migration, replication, and remote swap to enhance locality of accesses to shared pages. SciOS is implemented as a kernel module with interfaces to the VFS facility.

18.4.1 SciOS Memory Protocols

In SciOS, an application can request different memory protocols depending on the sharing pattern for a given file. The protocols range from a simple, fixed allocation of pages on a single node to a complex global memory management protocol. A protocol is selected with a file control call (`ioctl`). The different protocols are:

FIXED_INIT: Physical pages are always allocated on the node that has created the file. As remote writes have a lower latency than remote reads (see Chapter 3) the implementation of message passing interfaces and applications with multiple-producers/single-consumer patterns can benefit from using this memory protocol by optimizing the access patterns to use remote writes and local reads.

FIRST_TOUCH: Each physical page is allocated—and stays—on the node that first accessed the virtual page. This protocol is targeted at parallel applications with a mostly-write sharing pattern or little active sharing.

MIGRATION: Physical pages are allocated as in the FIRST_TOUCH protocol, but they may migrate individually between the nodes depending on the sharing pattern. A freeze/defrost mechanism [1] ensures that ping-pong is minimized when multiple nodes simultaneously access the same page, as a frozen page is not allowed to migrate. Sequential sharing patterns can benefit from this protocol.

MIG_REP: This is an extension of the MIGRATION protocol where physical pages are replicated when read by multiple nodes. Coherency mechanisms assure that the memory consistency model is respected. Sharing patterns with mostly-read shared data can use this protocol.

GLOBAL: This is similar to the MIG_REP protocol, but to globally minimize swap activity, decisions are coordinated with the virtual memory system and idle remote memory is used as swap space instead of local disk. This can benefit applications that need large amounts of memory.

SciOS uses relaxed memory consistency for the MIG_REP and GLOBAL protocols which replicate pages. The model is based on Lazy Release Consistency [10] which requires shared memory accesses to be synchronized. Instead of immediately invalidating all page copies when a replicated page is modified, they are not removed until the remote node enters a critical section. For correctly synchronized applications, this assures sequential consistency as expected by many applications.

18.4.2 Main File System Data Structures

SciOS has currently only implemented a root directory since the prototype is focused on the integration with the virtual memory system for implementing advanced memory management protocols, e.g., the GLOBAL protocol. All file system data is placed in shared memory which is accessible remotely through SCI. Linux does not distinguish between vnodes and inodes. Instead, both file system dependent and independent data are contained in a unified inode. In this section, an inode represents only the SciOS specific file information.

Both SciOS files and directories are represented by the same *file entry* data structure which is stored in a directory file (shown in Figure 18.4). It basically consists of a symbolic name and a pointer to an SciOS inode which holds information on the file size, memory protocol, protection information, etc. An *inode* holds a number of *page entries* which is sufficient for small files. The page entry data structure is shown in Table 18.4. For large files, an inode instead points to *page tables* which then contain the page entries (as in the virtual memory system).

`state`	Invalid, valid, frozen, or swapped to disk.
`lock`	Lock to assure exclusive access to the page.
`page_addr`	SCI address of the physical page or a swap disk address.
`copyset`	Indicates what nodes have a copy of the page.
`mapset`	indicates what nodes have the page remotely mapped.

Table 18.4. Fields in a page entry data structure which is contained in a SciOS inode or a page table.

The virtual memory pages for a memory-mapped SciOS file are uniquely identified in the cluster by a *pageid* which is a tuple (inode id, offset). On each node, a *map table* describes which local processes have mapped each SciOS file and at which virtual address. Given the map table and a pageid, each node can easily find the relevant Linux page table entries, e.g., when invalidating remote mappings, when migrating a page or when discarding remote page copies on a write access to the page.

18.4.3 The GLOBAL Memory Protocol

In an SCI cluster, a page can be transferred to/from a remote memory roughly 50 times faster than to/from a local disk. This makes it possible to implement highly efficient *global* memory management protocols that allow a node to use a remote node's memory as swap space instead of its local disk.

The GLOBAL memory protocol in SciOS [12] mixes remote mapping, migration, replication and distributed swapping of shared pages. A physical page is allocated on the first page fault ("first touch"). On subsequent page faults, three possibilities exist:

1. establish a mapping to the remote page (for frozen pages),
2. migrate the page to the local node, or
3. replicate the page to allow local accesses.

We first describe the cases when there is enough physical memory available. Then we describe the actions taken when a page must be swapped, i.e., when there is not enough physical memory. The actions performed on page faults are summarized in the decision tree shown in Figure 18.5.

Enough free physical memory. A page that is already present in the local physical memory, e.g., mapped by another process, is mapped directly in the process. If the page is placed on a local or remote disk, we allocate a new physical page and read the page from disk. If the page is in a remote memory, it is first checked if the page is or should be frozen. The decision to freeze a page is based on an *invalidation time* associated with each page. This value is updated, when the page is either migrated or has all its copies discarded on a write access. If the invalidation time of the page has been updated recently, the page is frozen. A frozen page will never be migrated, and is thus always mapped over the network.

A page that is not frozen can be either migrated or replicated. On a write access, the page is always *migrated* to the node of the writing process, any mappings of the page made by other nodes are invalidated, and the invalidation time is updated. On a read access the page can be *replicated*, if the page has not been modified recently. For replicated pages, the master copy and all the page copies are marked read-only to the virtual memory system. On a write access to a replicated page, a page fault is generated and all other page copies are discarded. Then the page is marked read/write, its

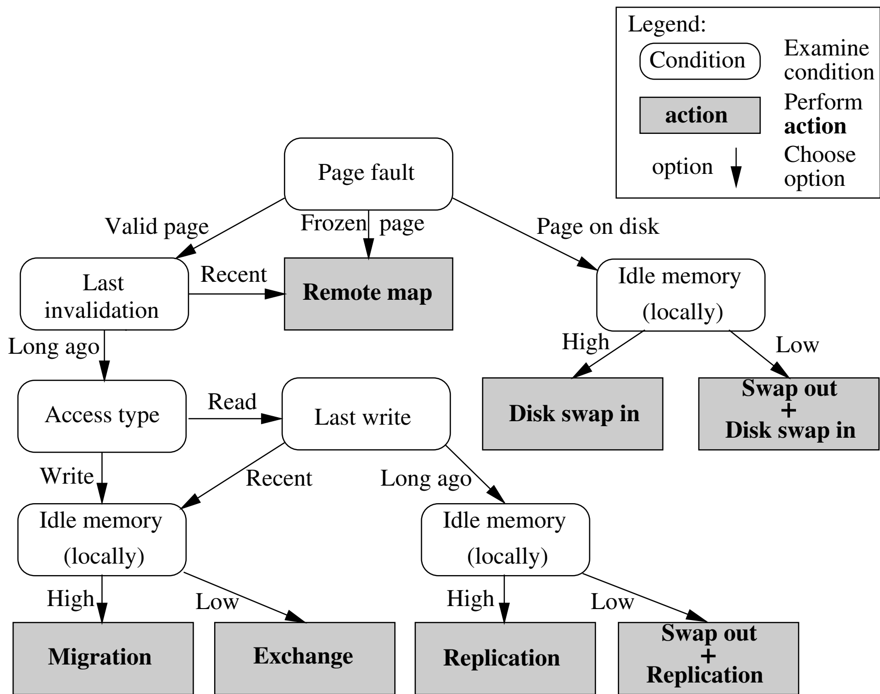


Fig. 18.5. Decision tree for handling page faults in SciOS

invalidation time is updated and the process can continue its modification of the page.

Not enough free physical memory. When it is not possible to allocate a new physical page, an old page must be swapped out first. The decision about where to place a page that needs to be swapped out depends on the type of page fault event that caused the swap out:

1. To make room for a migrating page from node Q, the node P needs to swap out its oldest page. This can be optimized by an exchange operation. Because a migration means that node Q frees a page, node P can simply exchange its oldest page with the migrating page from node Q.
2. A node P needs to free a page because it is fetching a page from a disk or replicating a remote page. If the oldest page on node P is not a replica or a frozen page, we ask the node with the globally oldest page, say node R, to swap it out. When R has finished, we copy the locally oldest page from node P to node R and then load the needed page from disk or replicate the page. (P and R can be the same node.)

In the second case, the node with the globally oldest pages is found by examining the weights computed by a daemon (see Section 18.4.4). The algorithm used when swapping out a page approximates a global LRU [3]. When

a node swaps out its oldest page and it is a replica, it can just be discarded if a copy of the page remains on another node. A frozen page can be migrated to a node that already maps it remotely.

18.4.4 Memory Protocol Implementation in Linux

The implementation of the different SciOS memory protocols are all based on the most complex protocol, GLOBAL. As an example, the MIG_REP protocol swaps pages to disk in contrast to GLOBAL which tries to place the page in an idle remote memory.

SciOS is implemented by providing the following:

Kernel module routines which can initialize and remove a Linux kernel module. The module consists of two layers on top of Dolphin's PCI-SCI driver, but it uses only a few low-level functions in the driver. The bottom layer extends the functionality of the driver with generic functions such as page allocation and mapping of remote pages. The top layer defines the SciOS file system type which is used for mounting.

Virtual file system operations that implement the mounting and unmounting of the file system and file operations such as `create`, `lookup`, `unlink` (see Table 18.2). We use the default Linux operations for `open` and `close`, and we have yet to implement many directory operations like `mkdir`.

A daemon, implemented as a kernel thread on each node, that periodically collects information on the file accesses of all processes mapping SciOS files. Frozen pages are defrosted after a certain time interval. Ages are computed for each physical page—mapped locally or not—and age statistics are distributed to the other nodes.

Virtual memory operations which, for each memory-mapped SciOS file, treat page faults (`nopage` and `wppage`) and swap events (`swapout`/`swapin`) originating from Linux's virtual memory system (see Table 18.3). The decision of the page fault handler for migration or replication may be based on information held on remote nodes. In those cases, communication is based on the PCI-SCI adapter's ability to generate remote interrupts, e.g., when invalidating remote mappings before migrating a page.

Synchronization primitives, implemented as `ioctl` calls, which are based on the PCI-SCI adapter's ability to perform atomic fetch-and-increment operations on remote memory. The primitives also interact with SciOS's consistency model. E.g., when acquiring a lock, all page copies that have been invalidated are removed, possibly stale cache blocks are invalidated, and prefetch buffers are emptied. On a lock release, SciOS notifies remote nodes about invalidated pages, flushes the processor's write buffers along with the PCI-SCI adapter's streams.

The implementation of GLOBAL depends much on the capabilities of the virtual memory system. The Linux virtual memory operations (see Table 18.3) specify the `wppage` operation which is called when a read-only page

is modified. In SciOS, it is used to detect writes to a replicated page. This operation was no longer implemented in Linux version 2.0 so we had to re-implement this functionality to have SciOS treat the event. The default action is for Linux 2.0 to do a copy-on-write which is not what SciOS needs in this case.

18.5 Related Work

SciOS draws on the research in multiple areas. Memory based file systems have traditionally been used to increase performance of access to temporary files. The earliest memory-based file systems used a fixed collection of physical pages as a block device (a RAM disk) with a conventional file system on top. This static allocation of physical pages may waste system resources. The Pageable Memory Based Filesystem [14] uses pageable memory to decrease the use of physical pages, but it still uses a block device interface to the memory resulting in a file system with a fixed size. The tmpfs [16] is a special purpose memory-based file system that makes use of the vnode/vfs framework. As tmpfs does not use a block device approach the file system implementation itself handles all memory administration in cooperation with the virtual memory system. This allows the file system to dynamically adjust its size according to usage. Metadata is kept in kernel memory, but all data is stored in pageable memory. SciOS extends this concept by providing a distributed memory-based file system.

The Global Memory Service (GMS) [3] implements a global swap mechanism based on approximate information about the cluster-wide state of the physical memory. GMS is implemented as a low-level operating system service that can be used by many types of clients, i.e., the paging system. Shared pages are replicated and are simply discarded if swapped out, like in SciOS. Consistency of shared pages is the responsibility of the GMS clients, but there are no mechanisms to handle consistency (e.g., there are no primitives to invalidate remote replicas). If a GMS client replicates pages itself, GMS will treat the replicas as distinct pages and two replicas can thus end up on the same node. The N-Chance forwarding algorithm [2] coordinates caches for a distributed file system. The node used for placing swapped out pages is randomly picked. The cache blocks are kept coherent using a single-writer/multiple readers protocol. But this can mean a ping-pong effect for pages that are write shared or are subject to false sharing effects.

The NUMA migration, freeze/defrost, and replication techniques used in SciOS are based on the results of PLATINUM [1], the studies by LaRowe *et al.* [13], and later CC-NUMA studies. Verghese *et al.* [18], like SciOS, take the amount of idle memory into consideration when replicating pages on a CC-NUMA architecture. Their system stops replicating pages when memory pressure is experienced on a node. With SciOS, we replicate pages even if it means that another page needs to be swapped out. Because remote

accesses are much more expensive on an SCI cluster than on CC-NUMA and because SciOS presently does not replicate at the cache block level for remotely mapped pages, we believe that page replication is an important optimization even under memory pressure. We do age replica pages faster than normal pages, thereby allowing the node to quickly free the physical page, e.g., for another replicated page.

The Cashmere-2L system [17] is implemented on a cluster of SMP nodes and uses the write-only based Memory Channel [5]. Memory coherency is maintained in hardware within each SMP node (first level) and is based on a home-based lazy release consistency model between nodes (second level). Cashmere-2L is, like TreadMarks [9], based on the twin/diffs technique which allows multiple writers to a shared page. Experiments show that the twin/diffing can perform better than “write doubling” on a one-level version of Cashmere. The write doubling technique tries to emulate a load/store interface on the write-only network not only making writes to a remote master copy, but as as well to a local copy. In SciOS , the twin/diffing can enhance performance—compared to a remote mapping to a frozen write-shared page—for applications with much cache locality for frozen pages. But the twin/diffing uses extra physical memory. We plan to study the twin/diffing technique together with the possibility enabling the processor caches for remote load/store accesses on the PCI-SCI adapter.

Ibel *et al.* have implemented a global address space at the user level using the Spilt-C language on an SCI cluster[8]. Because SciOS is implemented in the kernel, we can share the physical memory and remote mappings between applications/processes. SciOS can be used by many different languages and is not dependent on pointer indirections.

18.6 Summary and Conclusions

We have shown how a file system interface can be used as a low-level interface to SCI. The file system interface provides protection and symbolic naming of shared memory objects; both of them can be useful in large scale multi-user SCI clusters. Furthermore, memory mapped files provide efficient sharing of memory objects. All this is done using a stable and well-known interface. In addition, the interface must be augmented by synchronization mechanisms needed by parallel applications, e.g., with `ioctl` file control operations.

SCI can be integrated with the operating system kernel in a modular way across a wide range of operating systems using a virtual file system. The interface to the virtual memory system is not part of the virtual file system interface which can complicate the porting of complex file systems to other operating systems like Windows NT. The same issue exists for device driver interfaces to SCI.

Our prototype, SciOS, is an example of how a distributed shared virtual memory system with remote swapping facilities can be implemented in Li-

nux. SciOS is tightly integrated with the operating system, and the kernel interfaces are powerful enough to allow the implementation of even complex memory protocols. Even though a SciOS memory page can be swapped to disk, SciOS does not support files that survive system crashes. Because SciOS uses the file system interface, it is possible to extend the file system with non-volatile storage and reliability in a straightforward way. Such extensions will only require changes to the SciOS code, not the interface.

SciOS implements different memory protocols which can complicate the structuring of the file system. The latest development in *extensible file systems* [7] allows file system modules to extend the functionality of existing ones. Such functionality could be used to structure the implementation of the different SciOS memory protocols and to add new functions such as persistence.

Acknowledgements

Kåre Løchsen and Hugo Kohmann from Dolphin Interconnect Solutions (Norway) willingly granted us access to the driver source code for the PCI-SCI adapter and answered all of our questions. Roger Butenuth from Universität Paderborn (Germany) kindly gave us access to his Linux port of the PCI-SCI driver. Jean-Philippe Fassino assisted in the early implementation phase of SciOS. We are grateful for the discussions with members of INRIA's research groups who use SCI clusters. Eric Jul provided us with valuable comments and language corrections.

References

1. Alan Cox and Robert Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Computer Architecture*, December 1990.
2. Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, November 1994.
3. Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 201–212, December 1995.
4. Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.
5. Richard B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.
6. Berny Goodheart and James Cox. *The Magic Garden Explained: the Internals of UNIX System V Release 4, An Open Systems Design*. Prentice Hall, 1994.

7. John S. Heidemann and Gerald J. Popek. File-System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
8. Maximilian Ibel, Klaus E. Schauser, Chris J. Scheiman, and Manfred Weis. High-Performance Cluster Computing Using SCI. In *Hot Interconnects Symposium V*, August 1997.
9. Peter Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter USENIX Conference*, pages 115–132, January 1994.
10. Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, May 1992.
11. S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Summer Conference*, pages 238–247, 1986.
12. Povl T. Koch, Emmanuel Cecchet, and Xavier Rousset de Pina. Global Management of Coherent Shared Memory on an SCI Cluster. In *Proceedings of SCI Europe 98*, pages 51–57, Bordeaux (France), September 1998.
13. Richard P. Larowe Jr., Carla Schlatter Ellis, and Laurence S. Kaplan. The Robustness of NUMA Memory Management. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 137–151, October 1991.
14. Marshall K. McKusick, Michael K. Karels, and Keith Bostic. A Pageable Memory Based Filesystem. In *Proceedings of the Summer 1990 USENIX Technical Conference*, pages 137–143, June 1990.
15. Rajeev Nagar. *Windows NT File System Internals: A Developers Guide.* O'Reilly, 1997.
16. P. Snyder. tmpfs: A Virtual Memory File System. In *Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, pages 241–248, October 1990.
17. Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 170–183, October 1997.
18. Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, October 1996.

19. Programming SCI Clusters Using Parallel CORBA Objects

Thierry Priol[1], Christophe René[1], Guillaume Alléon[2]

[1] IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
email: {priol, crene}@irisa.fr
http://www.irisa.fr/
[2] Aerospatiale Joint Research Centre, 12 rue Pasteur, BP 76,
92152 Suresnes Cedex, France
email:Guillaume.Alleon@siege.aerospatiale.fr

19.1 Introduction

This chapter introduces a programming environment for SCI clusters that takes advantage of both parallel and distributed programming paradigms. It aims at helping programmers to design high performance applications based on the assembling of generic software components. This environment is based on CORBA (Common Object Request Broker Architecture), with our own extensions to support parallelism across several cluster nodes within a distributed system. Our contribution concerns extensions to support a new kind of object, which we call a parallel CORBA object (or parallel object), as well as the integration of message-passing paradigms, mainly MPI, within a parallel object. These extensions exploit as much as possible the functionality offered by CORBA and require few modifications to an available CORBA implementation. This paper reports on these extensions and the description of a runtime system, called *Cobra*, which provides resource allocation services for the execution of parallel objects.

The chapter is organized as follows. Section 19.2 discusses some issues related to parallel and distributed programming. Section 19.3 gives a short introduction to CORBA. Section 19.4 describes the concept of parallel CORBA objects. Section 19.5 introduces the *Cobra* runtime system for the execution of parallel objects. Section 19.6 presents a case study based on a signal processing application from Aerospatiale. Section 19.7 describes some related work which has some similarities with our work. Finally, Section 19.8 draws some conclusions and outlines perspectives of this work.

19.2 Parallel vs. Distributed Programming

Thanks to the rapid performance increase of today's computers, it can be now envisaged to couple several computationally intensive numerical codes to simulate more accurately complex physical phenomena. Due to both the increased complexity of these numerical codes and their future developments, a

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 333-348, 1999
© Springer-Verlag Berlin Heidelberg 1999

tight coupling of these codes cannot be envisaged. A loose coupling approach based on the use of several components offers a much more attractive solution. One can envisage to couple fluid and structure components or thermal and structure components. Other components can be devoted to pre-processing (data format conversion) or post-processing of data (visualization). Each of these components requires specific resources (computing power, graphics, specific I/O devices). A component which requires a huge amount of computing power can be parallelized so that it will be seen as a collection of processes to be run on a set of cluster nodes. Processes within a component have to exchange data and have to synchronize. Therefore, communication has to be performed at different levels: between components and within a component. However, the requirements for communication between or within a component are quite different. Within a component, since performance is critical, low-level message-passing is required, whereas between components, although performance is still required, modularity/interoperability and re-usability are necessary to develop cost effective applications using generic components.

However, until now, most programmers who are faced with the design of high-performance applications use low-level message-passing libraries such as MPI or PVM. Such libraries can be used for both coupling components and for handling communication among processes of a parallel component. It is obvious to say that this approach does not contribute to the design of applications using independent software components. Such communication libraries were developed for parallel programming; they do not offer the necessary support for designing components which can be reused by other applications.

Solutions already exist to decrease the design complexity of such applications. Distributed object-oriented technology is one of them. A complex application can be seen as a collection of objects which represent the components, running on different machines and interacting using remote object invocations. Emerging standards, such as CORBA, support the design of applications using independent software components through the use of CORBA objects. For the rest of the chapter, we will use the term object to name a CORBA object. CORBA is a distributed software platform which supports distributed object computing. However, exploitation of parallelism within such an object is restricted in a sense that it is limited to a single node within a cluster. CORBA implementations such as Orbix from Iona Technologies [8], allow the design of multi-threaded objects that can exploit several processors within a single SMP (Symmetric Multi-Processing) node. Such an SMP node cannot offer the large number of processors which is required for handling scientific applications in a reasonable time frame. However, the required number of processors is available at the cluster level where several dozens of machines are connected. Nevertheless, application designers have to deal “manually” with a large number of objects that have to be mapped onto different nodes of a cluster, and to distribute computations and data among these objects.

Therefore, either parallel and distributed programming environments have their limitations which do not allow the design of high performance applications using a set of reusable software components. The remaining part of this chapter introduces a programming environment which combines the advantages of both parallel and distributed programming.

19.3 An Overview of CORBA

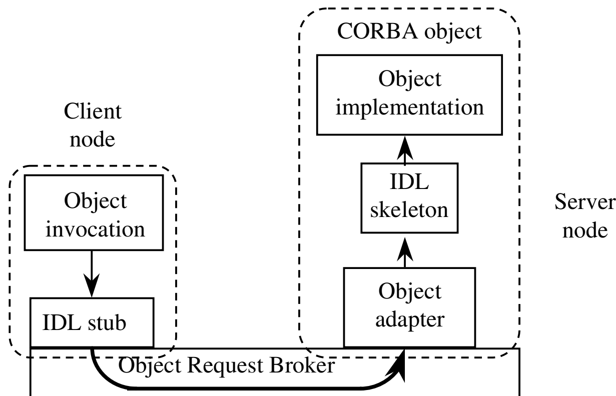


Fig. 19.1. CORBA system architecture

CORBA is a specification from the OMG (Object Management Group) [7] to support distributed object-oriented applications. An application based on CORBA can be seen as a collection of independent software components or CORBA objects. Remote method invocations are handled by an Object Request Broker (ORB) which provides a communication infrastructure independent of the underlying network. Within the ORB, several protocols exist to handle specific network technologies. The most important protocol is the IIOP (Internet Inter-ORB Protocol) which is used to support Ethernet-based networks. However, IIOP was designing for interoperability and thus offers limited performance. Fortunately, CORBA designers have provided the ESIOP (Environment-Specific Inter-ORB Protocol) which can handle other network technologies (SCI, for instance). An object interface is specified using the Interface Definition Language (IDL). An IDL file contains a list of operations for a given object that can be remotely invoked. Figure 19.1 provides a simplified view of the CORBA architecture. In this figure, an object located at the client side is bound to an implementation of an object located at the server side. When a client invokes an operation of the object, communication between the client and the server is performed through the ORB thanks to the IDL stub (client side) and the IDL skeleton (server side). The stub and

the skeleton are generated by an IDL compiler taking as input the IDL specification of the object. Since CORBA is independent of the language used for the object implementation, an IDL compiler may generate stubs for different languages (e.g., Java, C++, Smalltalk). An object can thus be implemented in C++ and called by a client implemented in Java. The following example shows a simple IDL interface:

```
interface myservice {
  void put(in double a);
  int put(out double a);
  double myop(inout long i, inout long j);
};
```

An interface corresponds to an object class and an operation to an object method. In this interface example, there are two operations associated with the interface. Each operation has a single parameter. An operation parameter is assigned a type which is similar to a C++ type (e.g., a scalar, an array). A keyword added just before the type specifies if the parameter is an input or an output parameter or both. IDL types are mapped to the language to be used at the server and the client side. IDL provides an interface inheritance mechanism so that services can be extended easily.

A CORBA-compliant system offers several services for the execution of distributed object-oriented applications. It provides object registration and activation through the use of repositories. Object registration consists of specifying a process that implements the object so that when an operation is called, the process is executed.

19.4 Parallel CORBA Objects

CORBA was not originally intended to support parallelism within an object. However, available CORBA implementations provide a multi-threading support for the implementation of objects. Such support is able to exploit several processors sharing a physical memory within a single computer. This level of parallelism does not require any modification to the CORBA specification since it concerns only the object implementation at the server side. Instead of having one thread assigned to an operation, it can be implemented using several threads. However, the sharing of a single physical memory does not allow a large number of processors since these could create bus and memory contention. One objective of our work is to exploit the several dozens of nodes available within a cluster to carry out a parallel execution of an object. To reach this objective, we introduce the concept of parallel CORBA object.

19.4.1 Execution Model

The concept of parallel objects relies on an SPMD (Single Program Multiple Data) execution model. A parallel object is a collection of identical objects

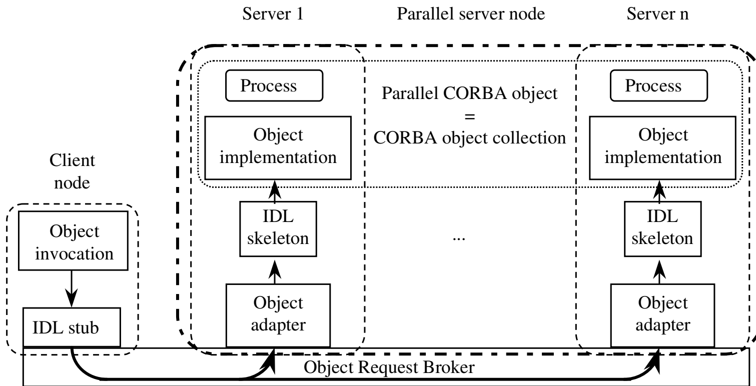


Fig. 19.2. Parallel CORBA object service execution model

having their own data, in compliance with the SPMD execution model. Figure 19.2 illustrates the concept of parallel objects. From the client side, there is no difference in calling a parallel object to calling a standard object. Parallelism is thus hidden to the user. When a call to an operation is performed by a client, the operation is executed by all objects belonging to the collection. The parallel execution is handled by the stub that was generated by an Extended-IDL compiler which is a modified version of the standard IDL compiler.

19.4.2 Extended-IDL

Like a standard object, a parallel object is associated with an interface which specifies the operations available. However, this interface is described using an IDL we extended to support parallelism. Extensions to the standard IDL aim at both specifying that an interface corresponds to a parallel object and at distributing parameter values among the collection of objects. Extended-IDL is the name of these extensions.

Specifying the degree of parallelism. The first IDL extension corresponds to the specification of the number of objects of the collection that will implement the parallel object. Modifications to the IDL language consist of adding two brackets to the IDL interface keyword. A parameter can be added within the two brackets to specify the number of objects belonging to the collection. This parameter can be a "*", which means that the number of objects belonging to the collection is not specified in the interface. An integer value or a function which determines the number of objects, is also valid. The following example illustrates the proposed extension:

```
interface[*] ComputeFEM {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err);
};
```

In the previous example, the number of objects will be determined at runtime depending on the available resources (i.e., the number of cluster nodes if we assume that each object of the collection is assigned to a single node). The implementation of a parallel CORBA object may require a given number of objects in the collection to be able to run correctly. The following code gives an example of a parallel object service which comprises four objects:

```
interface[4] ComputeFEM {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err);
};
```

Instead of giving a fixed number of objects in the collection, a function may be added to specify a valid number of objects in the collection. The following example illustrates this possibility. In this case, the number of objects in the collection may be only a power of two:

```
interface[n^2] ComputeFEM {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err);
};
```

IDL allows a new interface to inherit from an existing one. Parallel interfaces can do the same but with some restrictions. A parallel interface can inherit only from an existing parallel interface. Inheritance from a standard interface is prohibited. Moreover, inheritance is allowed only for parallel interfaces that can be implemented by a collection of objects with a corresponding number of objects. The following examples illustrate this restriction:

```
interface[*] MatrixComponent {
  void matrix_vector_mult(in double mat[100][100], in double v[100],
                          out double u[100]);
  void matrix_transpose(in double A[100][100],
                        out double B[100][100]);
};
interface[n^2] ComputeFEM : MatrixComponent {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err)
};
```

In this example, interface `ComputeFEM` inherits from interface `MatrixComponent`. The new interface has to be implemented using a collection having a square number of objects. In the following example, the inheritance is not valid:

```
interface[3] MatrixComponent {
  void matrix_vector_mult(in double mat[100][100],
                          in double v[100], out double u[100]);
  void matrix_transpose(in double A[100][100],
                        out double B[100][100]);
};
```

```
interface[n^2] ComputeFEM : MatrixComponent {
  void initFEM(in double mat[100][100], in double p);
  void doFEM(in long niter, out double err)
};
```

The Extended-IDL compiler will generate an error when compiling this specification. The intersection of the valid range of values of each inherited parallel interface and the new parallel interface must not be empty; otherwise the inheritance is not valid.

Specifying data distribution. Our second extension to the IDL language to support parallel objects concerns data distribution. Remember that the execution of a method on the client side will provoke the execution of the method on every object of the collection on the server side. Since each object of the collection performs a part of the work and has its own separate address space, we must envisage how to distribute the parameter values for each operation. We add new keywords to specify how to distribute the parameter values among the objects of the collection. The following paragraphs will explain how the data can be distributed for both `in`, `out` and `inout` modes depending on their type. A data-distribution extension of an IDL specification is allowed only for parameters of operations defined in a parallel interface.

The IDL language provides multidimensional fixed-size arrays which contain elements of the same type. Types can be either basic or constructed types. The size along each dimension has to be specified in the definition. We provide some extensions to allow the distribution of arrays among the objects of a collection. Data distribution specifications apply for `in`, `out` and `inout` modes. These data distribution specifications follow those already defined by HPF (High Performance Fortran). This design choice permits to map Extended-IDL to the HPF language in the future. It will be thus possible to implement a parallel object using HPF. Such a mapping could be based on the IDL to Fortran90 compiler which is being designed and implemented within the Esprit PACHA project. The following example gives a brief overview of the proposed extension:

```
interface[*] MatrixComponent {
  void matrix_vector_mult(in dist[BLOCK][*] double mat[100][100],
                          in double v[100],
                          out dist[CYCLIC] double u[100]);
};
```

This extension consists of adding the new keyword `dist`, which specifies how an array is distributed among the objects of the collection. The 2D array `mat` is distributed by blocks of rows. Since the parameter is assigned an `in` mode, each object of the collection will receive a block of rows instead of the whole array. A distributed array of a given IDL type is mapped to an unbounded sequence of this IDL type which has been extended to store information related to the distribution. An unbounded sequence offers the

advantage that its length is determined at runtime. Scattering of distributed arrays among the objects of a collection is performed by the stub generated by the Extended-IDL compiler. If the client is itself a parallel object, the stub is in charge of gathering data from client objects before sending them to the server objects with the correct distribution. In the previous example, the number of objects in the collection is not specified in the interface. Therefore, the number of elements assigned to a particular object can be known only at runtime. Parameter `v` does not have a data distribution specification so that each object receives the whole array. The last parameter `u` has an `out` mode assigned to it. Each object of the collection will send back to the client a part of the array. The code generated by the Extended-IDL compiler is in charge of gathering the data from the objects of the collection and to give them back to the client which invoked the operation. Gathering may include a redistribution of data if the client is itself a parallel object. As a matter of fact, distribution of variable `u` may not be identical at the client and the server side. At the client side, information related to the distribution is stored in the corresponding unbounded sequence structure. This information is accessed by the stub of the parallel object to redistribute data if necessary.

19.4.3 Implementation of Parallel CORBA Objects

As we have shown in the previous paragraphs, the code generated by the Extended-IDL compiler is in charge of managing the communication between a client that issues a request and a parallel object. The implementation relies on a new stub to be generated by the Extended-IDL compiler. After binding the client object to the parallel object, the client is able to send method invocations to the parallel object service. Modification of the ORB is not required since the CORBA specification provides a mechanism to issue multiple requests within a single call to the ORB. When an operation is invoked, a request is constructed containing the object references of all objects belonging to the collection as well as the name of the operation to be invoked. This request is then sent through the ORB which in turn will issue a request to each object to execute the operation.

19.5 The *Cobra* Runtime System

The *Cobra* runtime system provides resource allocation for the execution of parallel objects on SCI-based clusters. This runtime system is being developed with the Esprit PACHA R&D project. The project aims at building a parallel scalable computer system for high performance applications. This system includes both the development of hardware components and runtime systems. Emerging standards both in software, namely CORBA, and in hardware, namely SCI, are exploited to investigate the design and implementation

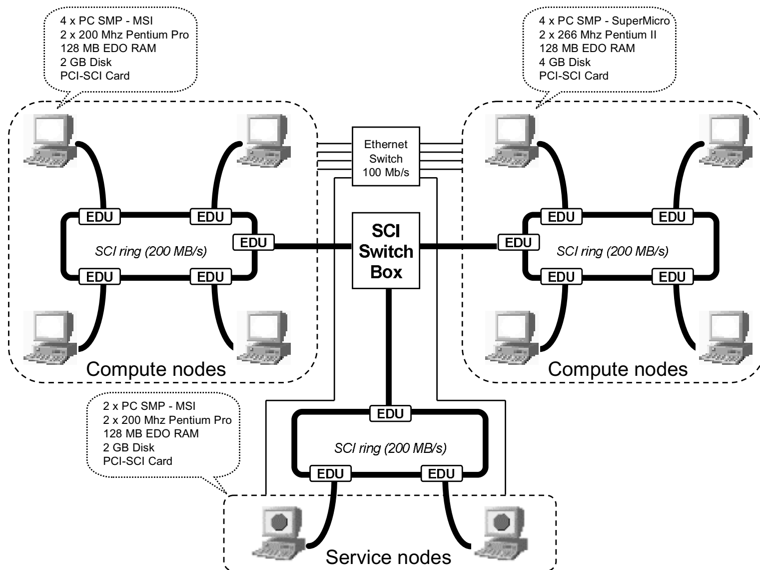


Fig. 19.3. The PACHA multiprocessors

of a full-featured CORBA-compliant software with minimum overhead. The *Cobra* runtime system is targeted to the PACHA multiprocessors, as shown in Figure 19.3. It is based on the clustering of PC systems using the PCI-SCI technology from Dolphin Interconnect Solutions. The machine is a set of three SCI ringlets connected together through an SCI switch. The first SCI ringlet contains two service nodes which act as the front-end of the PACHA machine. These two nodes run the *Cobra* runtime system for resource allocation. The two other SCI ringlets connect compute nodes. These compute nodes are allocated to users on demand by the *Cobra* runtime system for the execution of parallel objects. Resource allocation consists of providing cluster nodes and shared virtual memory regions for the execution of parallel objects.

19.5.1 *Cobra* Services

Cobra provides the concept of a virtual parallel machine which is associated with the execution of a parallel object. Allocating a virtual parallel machine consists of choosing a set of cluster nodes where objects of the collection will be mapped to for execution. Selection of nodes is performed statically since the PACHA multiprocessors act as a computational server. There are no other applications running simultaneously with parallel objects. Dynamic strategies could be added in the future to allow sharing of the machine by several user applications. *Cobra* offers services for the allocation of virtual shared memory regions. Such regions can be accessed simultaneously by several components

of the application. We think that the coupling of software components will need the exchange of unstructured data which cannot be passed efficiently between components through the ORB due to the cost of marshaling and de-marshaling data. The two resource allocation services are implemented using CORBA objects so that they are available from any machine within the cluster. Therefore, a client running somewhere in the network is able to allocate resources through CORBA, and once the resources have been allocated, a client can bind a parallel object to the virtual parallel machine which has been created.

Cobra provides basic services for parallel programming. Execution of objects belonging to a collection associated with a parallel object requires some basic functions such as identification and communication between objects. *Cobra* provides an application programming interface for these objects. This interfaces contains a set of C and Fortran77 functions for parallel programming such as synchronization between objects (barrier, lock), low-level message-passing and shared memory region management.

19.5.2 *Cobra* Software Architecture

Cobra has been designed for the PACHA multiprocessors and thus benefits from the SCI technology. *Cobra* can allocate both physical nodes and shared memory for the execution of parallel objects. Figure 19.4 shows the overall architecture of *Cobra*. *Cobra* is a set of three standard CORBA objects which run on the service nodes of the PACHA multiprocessors. Implementation of these services is carried out using either MICO [14], a freely available CORBA implementation from the University of Frankfurt, or ORBSCI, which is a CORBA implementation being implemented by Spacebel within the PACHA project.

The *AdminProcess* object provides services for the administration of the multiprocessors. The *RmProcess* gives a set of services for resource allocation, while *AppProcess* supports the execution of stand-alone applications. Since several service nodes are allowed by the runtime system, resource allocation tables are mapped onto SCI shared memory regions so that each service node is able to access the allocation tables. Running on the compute nodes, the *NodeProcess* object provides services to the *RmProcess* object for the execution of objects belonging to a collection. Accesses to these services are performed using either UNIX commands, specific APIs, or simply by using the IDL specification.

Administration service. The *AdminProcess* service administers the PACHA multiprocessors. It is mapped on a specific node of the machine which is called the administration node that acts also as a service node. There is always one such node in the PACHA multiprocessors. This service provides basic support for adding and removing nodes or changing the node state. For instance, at any time a compute node can be changed to a service node to

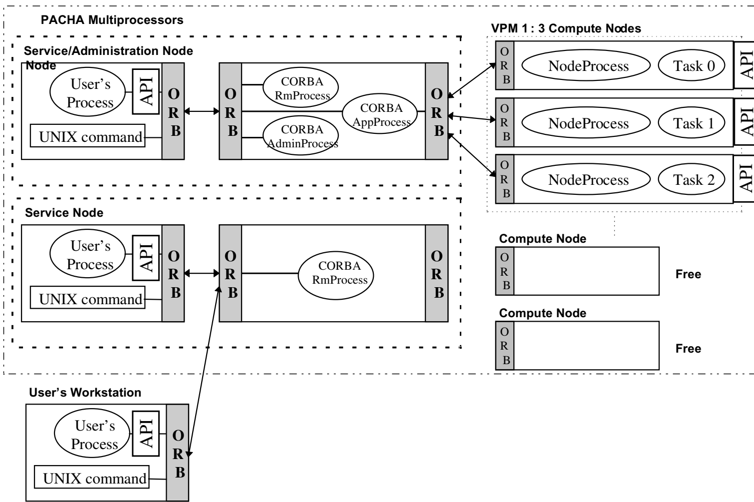


Fig. 19.4. The *Cobra* runtime system architecture

let more users have access to the PACHA multiprocessors. To protect the system, the *AdminProcesss* service can be executed only by a user who has administrator privileges. A list of users is maintained by the runtime system indicating if a user is an administrator or a standard user. For each node of the machine, the runtime system maintains a list of resources associated with that node (e.g., access to a fast network, frame buffer, number of processors). This information is used later when a set of nodes is allocated to a particular parallel object which may require specific resources for execution.

Resource management service. The *RmProcess* service provides resource management for standard users. This service is run on each service node of the PACHA multiprocessors. It manages resources such as compute nodes and shared memory regions provided by SCI. To let service nodes manage their own set of users concurrently and to avoid contention when the number of users increases, allocation tables are shared between service nodes. This sharing is performed using several SCI shared memory regions. Accesses to these tables, by both the *AdminProcess* and *RmProcess* services, are performed using critical sections to avoid any incoherent state.

The *Cobra* runtime system provides the concepts of a Virtual Parallel Machine (VPM) and Shared Memory Regions (SMR). A VPM is a set of compute nodes of the PACHA multiprocessors allocated by a user on demand. It is identified in the system by a name. A VPM is used for the execution of a parallel object. When executing a parallel object on a VPM, the runtime system creates as many objects as there are processors in the compute nodes of a VPM.

The second kind of resource managed by *Cobra* are shared memory regions. An SMR is identified by an unique name in the system. Shared memory

regions can be used in several ways. They permit objects which are executed on different nodes of a VPM to share data. An SMR is also a way to exchange information between parallel objects running on different VPMs either simultaneously or sequentially. Exporting or importing an SMR between VPMs is granted depending on access rights specified during the creation of the SMR. Data stored in an SMR is persistent. Shared memory regions can thus be seen as a data repository which can be used to avoid huge data transfers between objects. Concerning the implementation, SMRs managed by *Cobra* correspond either to SCI shared memory regions or Shared Virtual Memory (SVM) regions [13]. SVM is implemented using SCI as a communication layer, and it provides better performance since it offers page migration and replication. Coherence is enforced by a strong consistency protocol.

Application service. Although the *Cobra* runtime system was mainly designed to support execution of parallel objects, we provide a specific service, called *AppProcess*, for supporting stand-alone applications (which are not CORBA compliant). This service allows the loading and the monitoring of parallel application (either SPMD or MPMD) onto a VPM using a set of UNIX commands.

Application programming interface. The *Cobra* runtime system provides several application programming interfaces for both the C and Fortran languages. The first API supports the client side. It provides a function for each operation of the IDL specification of the *AdminProcess*, *RmProcess* and *AppProcess* services. At the server side, an API is provided for parallel programming. This API contains C and Fortran functions for node identification, low-level message passing, shared memory management and synchronization such as locks and barriers.

19.6 A Case Study: The IDAHO Application

The IDAHO application is being developed by the Aerospatiale Joint Research Centre for electromagnetic experimentation and simulation purposes. The IDAHO application is a set of tools to help the engineers in processing the data coming from experiments performed in an anechoic chamber. The electro-magnetic illumination is generated by a transmitter-receiver driven by the operator. The reduced model is placed on a rotating column in the anechoic chamber. For each angle of rotation (from 0 to 360 degrees) and each frequency (typically from 2 to 8 GHz), a complex number representing the reflected field is stored. This experimentation procedure can last up to 90 hours and generate up to 1 GByte of data.

The IDAHO application has been designed to adopt a client/server approach to let engineers process data remotely from their workstations. The most compute-intensive part of the application has been encapsulated in a

parallel object which contains several operations. Among them, the *normalization* operation computes the correction on the measured values to balance the noise effects of the anechoic chamber, using the measurements of the empty chamber and the measurements of a reference. We can then represent in 2D the reflected field for each rotation angle and each frequency. This representation is commonly named hologram. The *windowing normalization* operation is very similar to the previous one, except that it includes a window multiplication in the computation. This is used to focus the analysis around the measured object. The *ISAR* image computation is an operation to compute a 2D ISAR image, by making a 2D FFT on the data. With an ISAR image, we are able to locate on the object the reflecting points. The transverse response computation aims at calculating the 2D transverse response, by making a 1D FFT on one dimension of the measured data. With a transverse response, it is possible to follow a reflecting point while the object is rotating in the anechoic chamber. Each of these operations has been parallelized using the *Cobra* parallel application programming interface. The most complex task is the parallelization of the matrix transpose needed by the 2D FFT. The complexity is due to the limited size of shared memory regions provided by the SCI driver (512 kByte). Preliminary results have shown a speedup of three for the matrix transpose when running on a four-processor VPM.

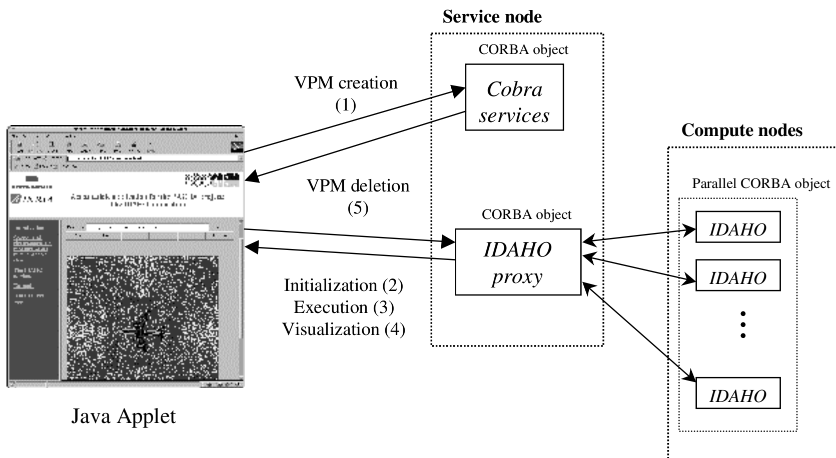


Fig. 19.5. The IDAHO client/server application

Visualization is performed using a graphical user interface which has been implemented in Java to be run on any machine in the network. A version of this interface has been designed and implemented as a Java applet. This applet is stored on a service node of the PACHA multiprocessor which acts as a Web server. Therefore, the IDAHO application can be executed from

any machine in the network having a Web browser supporting Java. Figure 19.5 shows the client/server overall architecture of the IDAHO application. In the first step (1), the applet connects to the *Cobra* services to allocate a VPM, then it connects to a proxy object which acts as a bridge between the applet and the parallel object. The proxy object has to be executed on the service node from which the applet was downloaded. The proxy object implements the same operations as the parallel object. The proxy object adds communication overhead but is required as a result of the security rules of the Java virtual machine. Once data has been sent to the parallel object (2), parallel execution starts (3) and an image is sent back to the applet (4) as a result of the execution. The final step (5) consists of releasing the VPM which was previously allocated.

19.7 Related Work

Several projects deal with environments for high-performance computing combining the benefits of distributed and parallel programming.

The RCS [1], NetSolve [4] and Ninf [15] projects provide an easy way to access linear algebra method libraries which run on remote supercomputers. Each method is described by a specific interface description language. Interface descriptions have been made for all methods of standard libraries (such as BLAS and LAPACK). Specific functions are provided for invoking methods of these libraries. Arguments of these functions specify method name and method arguments. These projects propose some mechanisms to manage load balancing on different supercomputers. One drawback of these environments is the difficulty for the user to add new functions to the libraries. Moreover, they are not compliant to relevant standards such as CORBA.

The Legion [5, 6] project aims at creating a world-wide environment for high-performance computing. A lot of principles of CORBA (such as heterogeneity management and object location) are provided by the Legion runtime system, although Legion is not CORBA-compliant. It manipulates parallel objects to obtain high performance. All these features are in common with our *Cobra* runtime system. However, Legion provides other services such as load balancing on different hosts, fault tolerance and security which are not present in *Cobra*. Furthermore, the Legion communication layer manages different networking technologies such as Ethernet and ATM.

The PARDIS [10, 11, 12] project proposes a solution very close to our project because it extends the CORBA object model to a parallel object model. A new IDL type is added: *dsequence* (for distributed sequence). It is a generalization of the CORBA sequence. This new sequence describes data type, data size, and how data must be distributed among objects. The PARDIS IDL compiler creates a new bind function, *spmd_bind*, which is added to the client's stub. Concurrent threads may use this function to make a collective request to a server. Therefore, this server has to reply only to one request.

For each operation listed in the interface description, the PARDIS IDL compiler adds a non-blocking method. Non-blocking functions may be executed concurrently even if they are called in a sequential order. `Out`-arguments of these functions are returned in *futures*. This idea results from the work on parallel C++. In PARDIS, distribution of objects is up to the programmer. This is the main difference from *Cobra*, for which a resource allocator is provided. Moreover, in *Cobra*, Extended-IDL allows to describe parallel services in more detail, such as the number of objects associated with a parallel object.

19.8 Conclusion and Perspectives

This chapter described the *Cobra* runtime system which provides a software environment for building high-performance applications using software components. *Cobra* is a set of CORBA services for the execution of CORBA parallel objects. A parallel object is a collection of standard CORBA objects. Its interface is described using an extension of IDL to manage data distribution among the objects of the collection. Current work now focuses on coupling numerical codes. Particular attention will be paid to the performance of the ORB which seems to be the most critical part of the software environment to get the desired performance. We are currently designing an efficient ORB for MICO, based on the Virtual Interface (VI) architecture.

References

1. P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. In *Proceedings of HPCN Europe '96*, volume 1067 of *LNCS*, pp. 662–667, Springer Verlag, 1996.
2. P. Beaugendre, T. Priol, G. Alleon, and D. Delavaux. A Client/Server Approach for HPC Applications within a Networking Environment. In *Proceedings of HPCN Europe '98*, volume 1401 of *LNCS*, pp. 518–525, Springer Verlag, 1998.
3. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
4. H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.
5. A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, University of Virginia, 1994.
6. A. S. Grimshaw, W. A. Wulf, and the Legion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 1(40):39–45, January 1997.
7. Object Management Group. The Common Object Request Broker: Architecture and Specification 2.1. August 1997.
8. C. Horn. The Orbix Architecture. Technical Report, IONA Technologies, August 1993.

9. Dolphin Interconnect Solutions. CluStar Interconnect Technology. White Paper, 1998.
10. K. Keahey. A Model of Interaction for Parallel Objects in a Heterogeneous Distributed Environment. Technical Report IUCS TR 467, Indiana University, September 1996.
11. K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. Technical Report IUCS TR 475, Indiana University, February 1997.
12. K. Keahey and D. Gannon. PARDIS: CORBA-based Architecture for Application-level Parallel Distributed Computation. In *Proceedings of Supercomputing '97*, November 1997.
13. K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.
14. A. Puder. The MICO CORBA Compliant System. *Dr. Dobb's Journal*, 291:44–57, November 1998.
15. M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure. In *Proceedings of HPCN Europe '97*, volume 1225 of *LNCS*, pages 491–502, Springer Verlag, 1997.

20. The MuSE Runtime System for SCI Clusters: A Flexible Combination of On-Stack Execution and Work Stealing

Markus Leberecht

LRR-TUM, Technische Universität München
email: Markus.Leberecht@in.tum.de
http://wwwbode.in.tum.de/~leberech/

20.1 Introduction

Beyond its high bandwidth and low latency properties, the Scalable Coherent Interface (SCI) technology offers capabilities for a shared-memory communication paradigm on distributed systems. In particular, networked workstations and PCs can basically be transformed into NUMA machines. As such, alternative execution models become possible that were otherwise infeasible on networks of workstations. With LANs, these normally rely on an inadequate communication infrastructure and they are fixed to the message-passing paradigm.

This chapter presents MuSE, the *Multithreaded Scheduling Environment*, a runtime system using a number of particular features unique to its SCI-based communication subsystem. MuSE provides an execution model that is a combination of dataflow-based self-scheduling, multi-threaded execution, and shared-memory communication with each of the three paradigms supporting the other two. Multithreaded execution is not only able to hide remote access latencies appearing during distributed shared memory communication, but it also helps to decrease dataflow-based scheduling overhead by avoiding too numerous and small active entities. dataflow in turn provides simple guidelines for self-controlled load balancing of threads, and profits from a shared memory-based communication paradigm in that a typical dataflow communication structure can be easily implemented by SCI's remote-memory store operations. MuSE's services cover the range common to most runtime systems, including machine virtualization, i.e., providing applications with a tractable machine model abstracted from the actual hardware, memory management, as well as load balancing and scheduling.

This chapter will focus on the methods and mechanisms used to implement MuSE's load-balancing capabilities. In order to do so on a distributed system, the migration of active entities is necessary. This degree of freedom is generally prohibitive of executing code on a local node with the highest possible efficiency. MuSE, however, attempts to enable load migration despite using highly efficient, stack-based sequential execution during phases without pending migration requests.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 349-364, 1999
© Springer-Verlag Berlin Heidelberg 1999

A good candidate for the inclusion of load migration methods is the function call mechanism, as here a natural border between active entities exists. During regular sequential execution, the most efficient method of calling a subroutine is to push the return address onto the program stack and to perform a branch operation. On return to the caller function, the previously stored return address, its *continuation*, can simply be retrieved by fetching it from the location pointed to by the stack pointer. A following stack pointer correction finishes the necessary actions. Using the stack for storing continuations only works because the program order for sequential programs can be mapped onto a sequential traversal of the tree of function invocations, the *call tree*. A parallel program execution, on the other hand, has to be mapped onto a parallel traversal of the call tree that is now often named *spawn tree*. A simple stack here does not suffice to hold the continuations (return addresses) of more than one nested execution as can be seen from Figure 20.1.

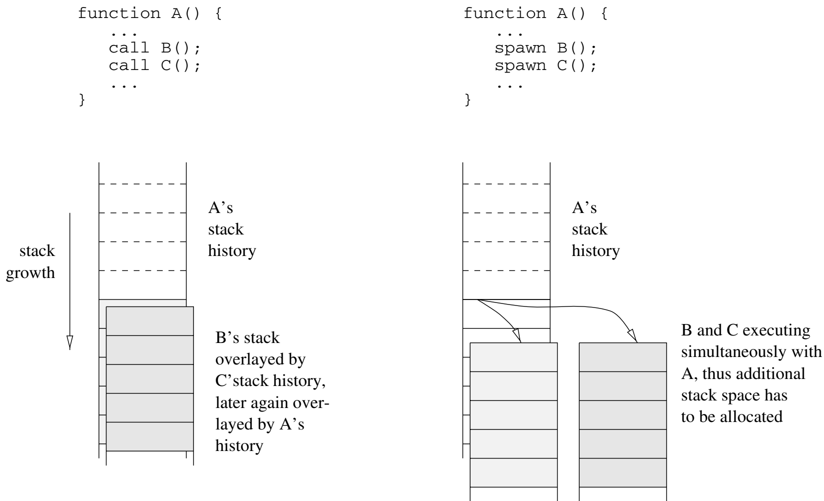


Fig. 20.1. Additional stack space is needed for parallel execution

Spawning, i.e., the allocation of function context memory from the heap, as a common runtime mechanism enables the migration of activities to underloaded nodes in a parallel or distributed system. Techniques of load balancing built on top of this feature are particularly important in the context of clusters of PCs with varying interactive background load.

This chapter first briefly introduces the SMiLE cluster, a network of PCs connected in a typical ringlet fashion by custom-developed SCI adapters. Some communication performance figures are given in order to motivate the need for low-overhead runtime mechanisms, in particular with respect to

work migration. The following section then describes MuSE, the Multithreaded Scheduling Environment, a scheduler prototype closely interfaced to the DSM-type communication of SCI with the ability to dynamically switch between on-stack and on-heap allocation of contexts. This property is utilized to implement a distributed work-stealing algorithm that achieves a significant improvement over runtime algorithms using purely heap-based execution on conventional networks of workstations. SCI's memory transactions are a crucial building block for this as they allow to use the memory transaction-based runtime mechanisms to be used in a distributed environment. A number of synthetic benchmarks are used to assess the performance of the proposed system. It can be concluded that speed-up scales well in the given small test environment. A comparison with already existing systems and an identification of further improvements finish the chapter.

20.2 The MuSE System

20.2.1 The SMiLE Cluster of PCs

The name SMiLE is an acronym for *Shared Memory in a LAN-like Environment* and represents the basic outline of the project. PCs are clustered with the *Scalable Coherent Interface (SCI)* interconnect. Utilizing SCI's built-in distributed shared memory model, the SMiLE project defines three major goals: development of hardware in order to utilize and push forward SCI technology, development of software in order to provide well-known programming interfaces on novel SCI-clustered systems, and development of concepts as well as methods for the efficient use of DSM systems in cluster environments. A good overview of the SMiLE project is given in [7].

The lack of commercially available SCI interface hardware for PCs during the initiation period of the SMiLE project led to the development of our custom PCI-SCI adapter card. Its primary goal is to serve as the basis of the SMiLE cluster by coupling the PC's I/O bus to the SCI interconnect. In order to facilitate new DSM tool concepts, additional monitoring functions can be easily integrated into this extensible system. The adapter is described in great detail in [1] and in Chapter 4.

The SMiLE configuration used in the context of this chapter features four Pentium PCs running at 133 MHz with 32 MByte of RAM and 2 GByte of external storage coupled by four SMiLE PCI-SCI adapters in a ring configuration. The operating system is Linux 2.0.33.

Active Messages 2.0 [9] have been implemented on SMiLE as a communication layer particularly suited to be a building block for distributed runtime system communication. Its zero-byte message latency was measured to be $14\,\mu s$ while a maximum throughput of 25 MByte/s can be reached with messages larger than 1 kByte.

20.2.2 The Multithreaded Scheduling Environment

The *Multithreaded Scheduling Environment (MuSE)* is a distributed runtime system specifically targeting the SMiLE platform. Programs running on MuSE are structured in a particular way in order to enable its load balancing strategy.

Two basic methods exist for this purpose: while in *work sharing* the load-generating nodes decide about the placement of new activities, *work stealing* lets under-loaded nodes request for new work. The *Cilk* project [2] highlighted that the technique of work stealing is well suited for load balancing on distributed systems such as clusters of workstations. Cilk implements an execution infrastructure consisting of C language extensions, a compiler, and a distributed scheduler. Programs are decomposed by the compiler into non-blocking fine-grain threads that can be run on any node of the system. This property is used to implement scheduler queues with a hierarchical structure. This so-called *ready dequeue* can hold the complete spawn tree of the application or parts of it. By migrating not yet executed thread contexts, the *closures*, to remote nodes, load balancing is achieved. By always choosing those closures closest to the root of the spawn tree, provably optimal on-line schedules with respect to time and space requirements can be achieved. However, in order to permit the application of this technique, Cilk requires every thread to be executed with a heap-allocated closure. This is necessary to avoid the situation depicted in Figure 20.1. However, heap allocation only pays off and justifies its overhead if the program execution actually uses the exhibited degree of freedom. For program parts that are not actually executed in parallel due to dynamic decisions or an excessive number of concurrent activities for the given number of processors, this means a potential source of inefficiency as unnecessary decomposition cost is paid.

MuSE in turn provides on-stack execution by default, avoiding this decomposition cost when it is not necessary. This is motivated by the attempt to profit from the high efficiency of sequential execution. In order to permit a Cilk-like work-stealing flexibility nonetheless, a different policy of spawning on-heap activities is required. The following structure of the runtime system and its active entities provides the basis for this.

Organization of Active Entities. MuSE programs are compiled from the dataflow language SISAL. In intermediate steps, appropriate dataflow graphs are generated which are subsequently converted into MuSE-compliant C code. The front end of the *Optimizing SISAL Compiler (OSC)* [3] combined with a custom-made code generator and the SMiLE system's `gcc` are utilized for this purpose.

MuSE Graphs. A MuSE *graph* is the C code representation of IF1/2 dataflow graphs and follows their semantics. IF1/2 descriptions are intermediate textual dataflow graph representations put out by OSC and are described in detail in [11]. IF1/2 implement a simple language for the description of hierarchical control and dataflow graphs. These graphs are represented in MuSE

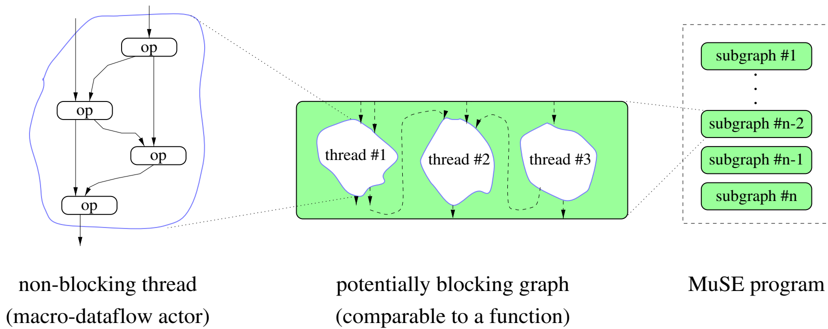


Fig. 20.2. MuSE program structure: non-blocking threads are combined into potentially blocking graphs, a collection of which forms the MuSE program

by regular C functions with an appropriate declaration of parameters as well as return values. Input data is brought into the graphs via regular C function parameters, as is return data. Access to these within the graph is performed via a context reference, in this case pointing to the appropriate stack location. Thus, sequential execution of MuSE graphs is able to utilize the simple and efficient on-stack parameter passing of conventional sequential languages.

For parallel execution, i.e., on-heap allocation of graph contexts, the same mechanism is used. By allocating heap storage and changing the context reference to heap storage, spawning is prepared whenever necessary, but transparently to the MuSE application code. The rationale behind the decision whether to either dynamically spawn or call a graph is explained below.

As spawning on a deeply nested level blocks at least one continuation within the caller until the callee has finished, heap context allocation will be performed for calling graphs on demand, resulting in lazily popping contexts off the stack.

In order to allow for blocking and to exploit intra-graph concurrency as well, graphs are subdivided into non-blocking activities called *threads*. Blocking by default has to occur at thread borders at which execution will later also resume. A graphical representation of graphs and their threads within a MuSE program is shown in Figure 20.2.

MuSE Threads. A MuSE *thread* is a non-blocking sequence of C statements that is executed strictly, i.e., once all input data is present, threads can be run to completion without interruption. Threads are declared within a MuSE graph with the `THREAD_START(thread_number)` statement and finished by the `THREAD_END` macro. These macros implement a simple guarded execution of the C statements forming the thread. The guard corresponds to a numerical synchronization counter responsible for implementing the dataflow firing rule.

MuSE threads are arranged in a simple as-soon-as-possible schedule within the graph as their underlying dataflow code is guaranteed by the compiler

to be non-cyclic. Thus, whenever no blocking occurs, a MuSE graph executes like a regular C function.

Contexts. As already pointed out, in the case of blocking, MuSE graphs need to save their local data for later invocations. By default, graph parameters are located on the stack as are graph-local variables. Thus, this data has to be removed from the stack and put into a heap-allocated structure, the graph *context*. A context has to reflect the data originally available on the stack during sequential execution and thus has to contain at least a pointer to the graph's C function, its input values as well as a container for the return value, some local variables, the threads' synchronization counters, and potentially the graph's continuation.

By choosing this form of implementing dataflow execution, MuSE effectively forms a two-level scheduling hierarchy. While the initial decision about which graph to execute is performed by the MuSE scheduler, the scheduling of threads within a graph is statically compiled in through the appropriate thread order. However, for nested sequential execution, the scheduler is only invoked once at the root of the call tree.

An advantage of this organization is the possibility to optimistically use calls instead of spawns to execute any MuSE graph. This means that only in those cases in which a spawn is occurring, the penalty for saving the context on the heap, memory allocation and copy operations, is paid.

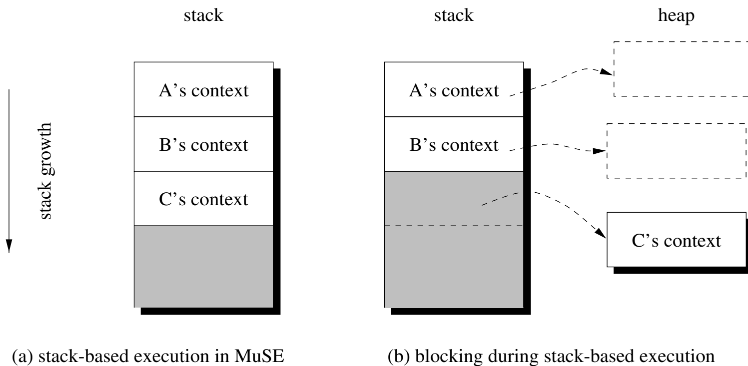


Fig. 20.3. Two cases during stack-based execution in MuSE: completion without blocking (a), or blocking at an arbitrary nesting level (b)

MuSE Stack Handling. MuSE by default treats execution as being sequential first, thus attempting to use on-stack execution as long as it is possible. Only when the actual need arises, heap storage is requested and execution can proceed concurrently as follows (see Figure 20.3):

1. In the non-blocking case (a), execution can be completely stack-based.

2. In (b), some graph C blocks. The blocking can occur several levels deeper on the stack than where the sequential starting procedure or graph A is located. As C blocks, its data is transferred to a heap-allocated context. C returns to B, telling it through its return value that it has blocked. For B this means that at least one of its threads, C's continuation, will not be executed. This results in subsequent blocking and heap allocation operations until A is reached and all intermediate levels have blocked. During this, however, all remaining work that is executable in the spawn subtree below A has already been performed.

The described situation points out MuSE's advantage: being stack-based by default, this execution model offers a basically unlimited stack space to each new graph call. Even in the case of a deeply nested blocking graph, all remaining enabled work can proceed sequentially. Only those graphs that really ever block need context space from the heap.

In detail, the mechanism works as follows:

– MuSE uses on-stack execution by default. Blocking is handled as described above and changes the execution mode for the blocked graphs to *on-heap*.
– On-stack execution does only involve the scheduler on the topmost level of the current spawn tree. Consequently, the polling operation within the main scheduler loop may be delayed. As this may hamper forward progress in the application and potentially starve other nodes due to not being serviced properly, the runtime system has to ensure that this does not happen. MuSE's strategy limits the total number of graph calls that are allowed to occur sequentially (`MAX_SEQUENTIAL`). Should no blocking operation occur during this time, the mode is nevertheless changed to on-heap execution in order to invoke the scheduler.

Load Balancing and Parallelism Generation in MuSE. On the thread level of execution, the dataflow firing rule provides the basis for an effective self-scheduling.

MuSE graphs are executed strictly, too. As a newly spawned context contains all input data and the graph's continuation, a context can basically be executed on any node. This property is used by the MuSE scheduler to distribute work across all participating nodes of a MuSE system.

MuSE Work Stealing. While MuSE threads correspond roughly to Cilk threads, their grouping into procedures or graphs has consequences regarding work stealing. In Cilk, thread closures are migrated regardless of them belonging to a particular procedure. This means that procedures do not exist for the work-stealing algorithm, and thus the execution of a procedure may be distributed over all nodes of the whole Cilk system.

In MuSE, threads do not have separate closures. Their parameters and synchronization counters are combined within the graph context. MuSE thus bases work stealing on contexts instead of single thread closures. It is hoped

that allocating a context once and performing a single migration per stolen graph, the slightly larger communication and memory management overhead of the larger graph context can be amortized. Additionally, data transports between threads can be performed in local memory in contrast to Cilk.

MuSE Queue Design. In Cilk, threads run to completion without being interrupted. The graphs in MuSE, however, can block. This means that a single ready queue does not suffice in MuSE. A graph that is currently not being executed can basically be in one of the three states `READY`, `BLOCKED`, and `UNBLOCKED` and is either placed in the *ready queue*, a *working queue*, or in a *networking pool*.

The ready queue has to service potential thief nodes as well as the local scheduler in the same way as in Cilk. Therefore, a structure comparable to the LIFO/FIFO token queue of the ADAM machine [5] was chosen, its functionality, however, is equivalent to Cilk's *ready dequeue*.

Communication. Communication across nodes in MuSE occurs in mainly three places:

– while passing return values,
– during compound data accesses, and
– during work stealing of ready contexts.

Simple SCI transactions do not completely cover the functionality required to implement all three cases mentioned above. Thus, they have to be complemented by actions taking place on the local and the remote node. For instance, passing of dataflow values requires updating of synchronization counters, while work stealing with its migration of complete contexts requires queue-handling capabilities connected to these communication operations.

In order to accomplish these tasks with the least overhead, MuSE bases all communication on the SMiLE Active Messages already mentioned in Section 20.2.1. This is in accordance with other, similar projects. Cilk as well as the Concert system [10] all rely on comparable messaging layers. Active Messages (AMs) have distinct advantages in this case over other communication techniques:

1. The definition of request/response pairs of handler functions enables the emulation of most communication paradigms through Active Messages. For MuSE, the AM handler functions are structured such that they represent memory-oriented transactions operating directly with dataflow tokens on graph context data. Basing this further on SCI, only a small amount of additional overhead is introduced for the added functionality and the convenient addressing scheme.
2. The handler concept of AMs allows to define sender-initiated actions at the receiver without the receiver side being actively involved apart from a regular polling operation.

3. Due to the zero-copy implementation on the SMiLE cluster, Active Messages actually offer the least overhead possible in extending SCI's memory transactions to a complete messaging layer, as can be seen from its performance figures [4].

Polling Versus Interrupts. Technical implementations of SCI-generated remote interrupts unfortunately exhibit high notification latencies, as documented in [8] for the Solaris operating system and Dolphin's SBus2-SCI adapter. The reason for this lies in context switch costs and signal-delivery times within the operating system. Thus, in order to avoid these, MuSE has to deal with repeatedly polling for incoming messages. In its current implementation, MuSE therefore checks the AM layer once during every scheduler invocation.

20.3 Experimental Evaluation

In order to assess MuSE's capabilities, first its single-node scheduler performance is characterized by timing runtime system services that are typically executed repeatedly during the scheduler loop.

Following this, experiments on the four-node SMiLE system were performed testing load-balancing properties under the light of SCI's high raw communication performance. The question is answered which run length SCI effectively allows for the migration of graph contexts without sacrificing parallel performance and the scalability of the presented runtime system concept.

Additionally, the evaluation is completed by a test of MuSE's load-balancing behavior under heterogeneous background loads.

20.3.1 Basic Runtime System Performance

Table 20.1 summarizes the performance of several basic runtime services of MuSE on the SMiLE cluster that are explained in more detail below:

t_{AM_Poll} is the time required for the Active Messages poll operation when no incoming messages are pending. This effort is spent during every scheduler invocation.

t_{get_graph} represents the time typically spent for finding a graph in the queues to be executed next.

$t_{exec,best}$ is the duration of an invocation of a graph by the scheduler whenever the context data is cached within the processor's first-level and second-level caches.

$t_{exec,worst}$ in contrast denotes the same time spent with all context data being non-cached in the processor caches (e.g., due to too large working sets).

t_{call} is the amount of time being spent for the on-stack calling mechanism.

t_{spawn} on the other hand represents the amount of time required for spawning a graph, i.e., allocating new context memory from the heap, initializing it, and placing it into the appropriate queue.

t_{return_data} summarizes the time of the operations necessary for passing return values on the same node whenever this cannot be done on stack.

t_{queue_mgmt} finally represents such actions as removing a context from one queue and placing it into another.

The most important result of Table 20.1 is the fact that the on-stack call overhead is smaller than the spawn overhead by more than an order of magnitude.

Unfortunately, the uncached execution of an empty graph points to a potential source of inefficiency of the current MuSE implementation: taking more than 20 μs doing no actual application processing may be just too high for significantly fine-grained parallel programs.

Function	Symbol	Time [μs]
empty poll operation	t_{AM_Poll}	0.9
obtain graph reference	t_{get_graph}	0.25
empty graph execution (cached)	$t_{exec,best}$	0.92
graph call overhead	t_{call}	0.26
graph spawn overhead	t_{spawn}	3.9
empty graph execution (non-cached)	$t_{exec,worst}$	21.9
local return-data passing	t_{return_data}	5.1
queue management	t_{queue_mgmt}	2.95

Table 20.1. Performance of various runtime system services

20.3.2 Load Balancing and Parallelism Generation

Load balancing. `knary`(k, m, i) is a synthetic test program that dynamically builds a spawn tree. Three parameters define its behavior, of which the spawn degree k describes the number of subgraphs being spawned in each graph, the tree height m specifies the maximum recursion level to which spawning occurs, and i is proportional to the number of idle operations performed in each graph before the spawn operation gets executed.

The first experiment consists of running `knary` on MuSE in a pure *on-heap* execution mode. The work-stealing mechanism performs load balancing among the participating nodes. This mode of operation is essentially equivalent to Cilk's. The case with $k = 4$ and $m = 10$ showed the most representative behavior and is thus used in the remainder of this section.

Figure 20.4 displays the speed-up curves related to the respective single-node execution time. The result is as expected: work stealing with per-default on-heap allocation of contexts achieves good speed-up ratios. MuSE thus

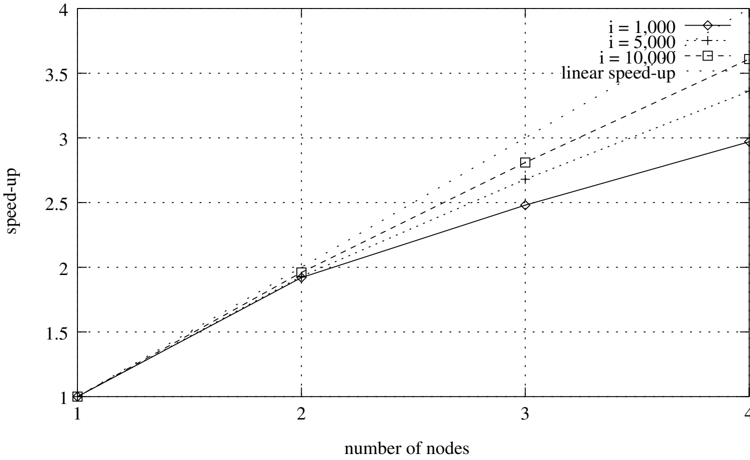


Fig. 20.4. Speed-up ratios of `knary`(4, 10, 1000), `knary`(4, 10, 5000), and `knary`(4, 10, 10000) with sequential execution times of 20.6 s, 43.0 s, and 73.1 s respectively

conserves the positive Cilk-like properties. The sub-linear speed-up can be explained by the significant costs of work stealing. The work-stealing latency is dominated by the time required for migrating a graph context from the victim to the thief node. Sending this 364 byte-sized chunk of memory is performed via a medium-length Active Message and takes approximately $t_{migrate} \simeq 120\,\mu s$ using the SMiLE AMs, representing a throughput of about 3.3 MByte/s for this message size. Only a few microseconds have to be accounted for the ready queue management on both sides and for the sending of the return data token. The experimental data supports this reasoning. As soon as the average execution time of a graph grows beyond the typical work-stealing latency—a value of $i = 5000$ represents an approximate graph runtime of $123\,\mu s$—work stealing becomes increasingly beneficial.

Load balancing with dynamic on-stack execution. It is expected that resorting to stack-based execution for mostly sequential parts of a program will help to speed up a given application as the overhead for spawning in relation to the actual computation time shrinks. Switching these modes is based upon the guidelines presented in Section 20.2.2.

Again, `knary`(4, 10, 1000) was used to derive reasonable numbers for both limits. Figure 20.5 displays the speed-up ratios for three different cases of `MAX_SEQUENTIAL`.

Absolute run times of the purely heap-based and the dynamically switched execution modes can be measured to drop by a factor of 2.8 to 3.4, regardless of the `MAX_SEQUENTIAL` setting used. Hence, dynamic switching between the stack-based and the heap-based execution modes obviously of-

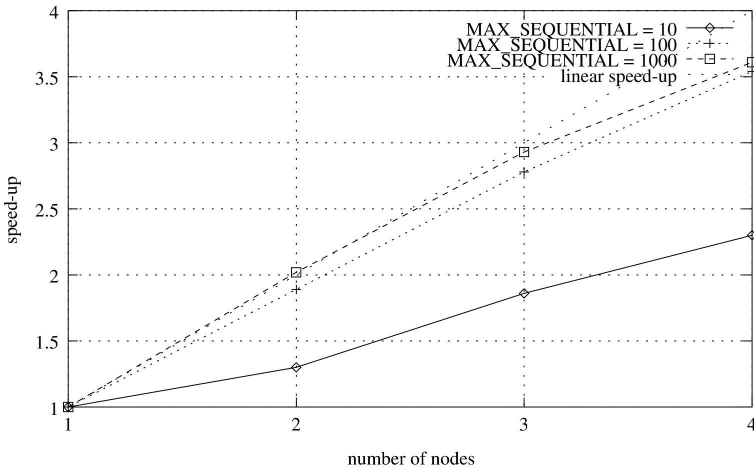


Fig. 20.5. Speed-up of `knary(4, 10, 1000)` for different values of `MAX_SEQUENTIAL`. Sequential runtime was $t = 7.3\,s$

fers fundamental improvements over the purely off-stack execution of the previous experiments.

In the single-node case, MuSE takes care of executing the application purely on-stack, requiring only the initial graph plus its first level of four calls to be executed with a spawned context. Its overhead is easily amortized over the $\sum_{i=1}^{9} 4^i = 349524$ total graph calls in this program and thus represents the truly fastest single-node implementation. In contrast to this, the single-node runs of the previous experiments had to pay all the unnecessary spawn overheads.

Increasing `MAX_SEQUENTIAL` from 10 to higher values also improves speed-up ratios, as can be seen from Figure 20.5. `MAX_SEQUENTIAL` $= 100$ was chosen as a compromise value since it realizes good speed-up while at the same time keeping the polling latency low. For instance, an average graph runtime of $2\,\mu s$ would yield a maximum of $200\,\mu s$ between two polls invoked by the scheduler. Higher values for `MAX_SEQUENTIAL` can still improve the speed-up somewhat, yet also increase the polling latency.

Load balancing under heterogeneous loads. In order to assess MuSE's capability of dealing with unevenly distributed background load while retaining its ability of switching execution modes dynamically, artificial load was systematically placed on each SMiLE node in the form of separate Linux processes in endless loops. Assuming a normalized processing throughput of $Y_i = \frac{1}{p}$ for each node with p equally active processes on the node, the maximum ideal speed-up for n nodes would be

$$S_{max} = \sum_{i=1}^{n} Y_i. \tag{20.1}$$

Thus, for a single background load on one node and two otherwise unloaded nodes, MuSE execution can exploit a cumulated throughput of $\sum Y_i = \frac{1}{2} + 1 + 1 = 2.5$ and therefore hope for a maximum speed-up of $S_{max} = 2.5$.

Again, `knary`(4, 10, 1000) served as the test case. Table 20.2 summarizes the experiments with up to four background loads.

nodes	wall-clock time [s]	speed-up	max. ideal speed-up	utilized max. performance
1 background load				
1	14.05	0.52	0.5	> 100 %
2	5.33	1.37	1.5	91.3 %
3	3.25	2.25	2.5	90.0 %
4	2.47	2.96	3.5	84.4 %
2 background loads				
2	7.70	0.95	1	95.0 %
3	4.26	1.71	2	85.7 %
4	3.42	2.13	3	71.2 %
3 background loads				
3	5.71	1.28	1.5	85.3 %
4	4.05	1.80	2.5	72.1 %
4 background loads				
4	4.84	1.51	2	75.4 %

Table 20.2. Absolute run times and speed-up values for `knary`(4, 10, 1000) and up to four background loads

It is clearly visible that MuSE is able to balance the load even when the processing performance of the nodes is uneven. However, when the number of background processes increases, less and less of the offered throughput can be utilized. This can be attributed to the uncoupled Linux schedulers which are bound to impede communication among the nodes more than necessary: while node A sends a request to node B, the latter is no longer being executed, forcing node A to wait for the next time slice of B.

Running more realistic SISAL applications unfortunately requires further tuning of the current prototypical and mostly C macro-based runtime system mechanisms. Additionally, an efficient compound data handling facility is important in order to allow MuSE graphs to perform array accesses with the same references regardless of the node on which they are executed. Unfortunately, the prevailing implementation of MuSE's compound data handling suffers from significant overheads that prevent actual application speed-up. This work therefore only covers synthetic benchmarks that nevertheless represent typical model cases.

20.4 Related Work and Conclusion

The MuSE and Cilk execution models seem similar on the first glance as both employ fine-grained non-blocking threads, both use hierarchical work stealing, and both employ dataflow mechanisms for easy self-scheduling. However, they differ in a specific point. In general, load balancing actions usually introduce overhead due to two tasks: *decomposition costs* for setting up independent activities and *load migration costs* for transporting an activity to a different node. While Cilk couples both actions and exhibits their costs even when decomposition might not be necessary, MuSE attempts to keep them apart.

While in Cilk on-heap execution is the only and default execution mode, MuSE's default paradigm lets execution run on the stack and only switches back to on-heap execution when demanded. In other words, it means that decomposition cost for parallel execution in Cilk is implicitly hidden in the execution model and proportional to the number of Cilk threads being spawned—even for the sequential case in which this would not be necessary—while MuSE's costs for generating parallelism are made explicit by mode changes due to work-stealing or fairness demands. This also means that Cilk's good speed-up values are achieved partly due to a non-optimal sequential version while MuSE exhibits the fairer comparison: decomposition into concurrent active entities, the MuSE graphs with their contexts, only occurs when it is really necessary. In Cilk, only load migration cost has to be offset by the performance improvement while MuSE has to offset both costs.

Runtime systems suitable for fine-grained parallelism have to be efficient when supporting sequential as well as parallel programs. They thus have to overcome the difficulty of trading off efficient on-stack execution modes against more flexible but less lightweight on-heap execution modes. Several methods for this have been presented so far:

- The *Lazy Threads* project [6] attempts to amortize heap allocation overhead by utilizing *stacklets*, independent and fixed-size chunks of heap space serving as stack for intermediate sequential execution. On stacklet overrun or a spawn operation, new stacklets are allocated, thus requiring static knowledge about when to spawn and when to call.
  In Lazy Threads, the decision about spawning and calling is non-reversible and has to be made at compile-time. MuSE in contrast offers a truly transparent model with the possibility to change execution modes at any time during the application's run.
- The *StackThreads* approach [12] defaults to sequential on-stack execution. In case of a spawn operation, a *reply box* in the calling thread serves as the point of synchronization for the spawned thread and its caller, blocking the caller until the spawned thread has completed.
  MuSE advances beyond StackThreads by not necessarily blocking a higher level activity as soon as a procedure has been spawned. Intra-graph parallelism through threads ensures that all ready execution can proceed on

the stack while in StackThreads a single blocking operation always blocks an entire call tree.

– In the *Concert* system [10], differently compiled versions of the same code supporting calling as well as spawning are utilized dynamically.
  In contrast to Concert, MuSE's method requires only a single object code, avoiding the storage overhead of multiple versions.

MuSE thus offers distinct advantages over these described runtime systems. Unfortunately, its prototype implementation suffers from a number of inefficiencies. C macros that are used to implement certain runtime system functions and to enhance manual readability are not flexible enough for the presented purpose. They do not perform sufficiently well to allow for even more fine-grained parallelism and realistic dataflow applications. Although this is no actual surprise, the macros behaved interestingly well as a testbed for the presented mechanisms. Machine-level implementations of the runtime mechanisms, however, seem more appropriate as well as using more compiler knowledge to resort to less general and more optimized functionality whenever possible.

In essence, the MuSE system shows that although high-performance networks such as SCI pose significant challenges for the runtime environment of clustered architecture, they can nevertheless be dealt with in a flexible and transparent way for application and runtime services, thanks to SCI's memory transaction-oriented nature.

References

1. G. Acher, H. Hellwagner, W. Karl, and M. Leberecht. A PCI-SCI Bridge for Building a PC Cluster with Distributed Shared Memory. In *Proc. 6th International Workshop on SCI-Based High-Performance Low-Cost Computing*, SCIzzL, Santa Clara, CA, September 1996.
2. R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proc. 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Nov. 1994.
3. D. C. Cann. *The Optimizing SISAL Compiler: Version 12.0.* Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April 1992.
4. M. Eberl, H. Hellwagner, W. Karl, M. Leberecht, and J. Weidendorfer. Fast Communication Libraries on an SCI Cluster. In A. Reinefeld and H. Hellwagner, editors, *Scalable Coherent Interface: Technology and Applications (Proc. SCI-Europe '98)*, pages 165–175, Cheshire Henbury, September 1998.
5. P. Färber. *Execution Architecture of the Multithreaded ADAM Prototype.* PhD thesis, Eidgenössische Technische Hochschule, Zürich, Switzerland, 1996.
6. S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.
7. H. Hellwagner, W. Karl, and M. Leberecht. Enabling a PC Cluster for High-Performance Computing. *SPEEDUP Journal*, June 1997.

8. M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. High-Performance Cluster Computing Using Scalable Coherent Interface. In *Proc. 7th Workshop on Low-Cost/High-Performance Computing (SCIzzL-7)*, SCIzzL, Santa Clara, CA, March 1997.
9. A. M. Mainwaring and D. E. Culler. *Active Messages: Organization and Applications Programming Interface*. Tech. Report, Computer Science Division, University of California at Berkeley, 1995. `http://now.cs.berkeley.edu/Papers/Papers/am-spec.ps`.
10. J. Plevyak, V. Karamcheti, X. Zhang, and A. Chien. A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers. In *Proc. 1995 ACM/IEEE Supercomputing Conference*, ACM/IEEE, December 1995.
11. S. Skedzielewski and J. Glauert. *IF1 - An Intermediate Form for Applicative Languages*. Technical Report TR M-170, Lawrence Livermore National Laboratory, July 1985.
12. K. Taura, S. Matsuoka, and A. Yonezawa. StackThreads: An Abstract Machine for Scheduling Fine-Grain Threads on Stock CPUs. In T. Ito and A. Yonezawa, editors, *Proc. International Workshop on the Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes of Computer Science*, pages 121–136, Springer Verlag, November 1994.

Part VII

Benchmark Results and Application Experiences

How much does a parallel application benefit by using SCI rather than a simple LAN? Is it really feasible to turn a collection of standard PCs into a dedicated high-performance computer just by interconnecting them with SCI? And how about the usability of such systems? Can they be reliably deployed in harsh multi-user environments, or are they restricted to specific applications?

This part has three chapters dealing with questions like these—and three different answers:

- the first one from a computer scientist's point of view, who is trained to assess a system's performance with a set of carefully selected synthetic benchmarks;
- the second one from an end-user's point of view, who seeks maximum performance for his/her specific application; and
- the third one from a physicist's point of view, who poses extremely high demands on the communication system needed for the real-time filtering of data gathered in high-energy physics experiments.

In Chapter 21, researchers from Paderborn Center for Parallel Computing (PC^2) report on their experiences gained in the design, installation, and operation of two very large SCI clusters. With 64 and 192 Intel Pentium II processors, respectively, these systems are not treated as exotic systems for selected users, but they are operated in the hard environment of everyday general-purpose multi-user computing. In their paper, the authors from Paderborn report on the decisions they took in the design of the system architecture, on their practical experiences and on performance results obtained with benchmark programs like Linpack, FFT, and HINT, and with real applications from three different domains.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 365-366, 1999
© Springer-Verlag Berlin Heidelberg 1999

Chapter 22 focuses on experiences gained at RWTH Aachen in porting and benchmarking a complex molecular dynamics code on an SCI cluster with eight Intel PentiumPro machines. This project is seen as part of a larger effort in studying parallelization methods for non-uniform memory access (NUMA) systems. In this specific case, the important molecular dynamics code GROMOS-87 has been ported to an SCI cluster using the shared memory interface *SMI* developed by the researchers in Aachen (see Chapter 16).

Chapter 23 finally shows that there are more uses to SCI than just general purpose cluster computing. Researchers at CERN, Rutherford Appleton Lab, Argonne National Lab, and the University of Manchester are currently testing the usability of SCI in the context of high-energy physics experiments. In the ATLAS experiment, for example, SCI is planned to be used in the second-level trigger in the Large Hadron Collider (LHC) at CERN, which will allow scientists to penetrate deeper into the structure of matter than has previously been possible. Based on their current 16-node dual-Pentium system and a 16-port SCI switch, the CERN researchers prepare a technical proposal for the large ATLAS high-level trigger system.

21. Large-Scale SCI Clusters in Practice: Architecture and Performance

Jens Simon[1], Alexander Reinefeld[1], Oliver Heinz[2]

[1] Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany
email: simon@zib.de, ar@zib.de
http://www.zib.de/

[2] Paderborn Center for Parallel Computing, D-33102 Paderborn, Germany
email: heinz@upb.de
http://www.upb.de/pc2/

21.1 Introduction

The *Paderborn Scalable Compute Server (PSC Server)* is a prototype of a commercial product that has been installed in 1997/98 by a vendor consortium consisting of Siemens AG and Scali Computer AS at the *Paderborn Center for Parallel Computing*. It builds on the COTS concept, that is, on the use of common off-the-shelf technology.

After extensive stability and performance tests, PSC Servers of various sizes are now marketed by Siemens under the brand name *hpcLine*. They provide truly scalable computing power ranging from a few GFlop/s with small clusters up to more than a hundred GFlop/s on high-end systems with several hundred Intel processors.

In this chapter, we report on the experiences gained with two large-scale installations with 64 and 192 Intel Pentium II processors, respectively. We give results on low-level benchmarks as well as real-world applications. While the results are still preliminary (because the system is subject to further optimizations) they provide valuable insights in the characteristics and useability of large dedicated compute clusters with standard off-the-shelf PC-technology and SCI interconnect.

21.2 PSC System Architecture

Apart from several small systems with various hardware components for testing purposes, we operate two large-scale SCI compute servers at our institute, both containing Siemens multiprocessor PCs and Dolphin PCI/SCI adapter cards. Their different hardware specification (300 MHz versus 450 MHz Intel Pentium II) reflects the technological advance between their installation dates in 1997 and 1998 respectively. But also their physical appea-

[†] The work presented in this chapter was done while all three authors were at Paderborn Center for Parallel Computing, http://www.upb.de/pc2

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 367-381, 1999
© Springer-Verlag Berlin Heidelberg 1999

Fig. 21.1. PSC-64 front- and backside view

Fig. 21.2. PSC-192, also available as Siemens hpc*L*ine

rance is quite different: The smaller system contains standard PC boxes that are simply stacked on top of each other in four cabinets (Figure 21.1), while the larger system contains just the PC motherboards without boxes in six cabinets (Figure 21.2). This was necessary, because the SCI technology used in our system does not allow for cable lengths of more than a few meters. Also, by removing unnecessary PC components (such as CD drives, diskettes, graphic cards, etc.) we were able to reduce the cooling effort and power consumption.

21.2.1 Node Configuration

PSC-64 contains a total of 32 Siemens Celsius PCs, each of them with a dual Intel Pentium II motherboard with BX chip-set, 256 MByte SDRAM, a local disk drive and an LC2-based PCI/SCI interface card. The 300 MHz

	PSC-64	PSC-192
node	32 Siemens Celsius	96 Siemens Primergy
processor	64 Intel Pentium II	192 Intel Pentium II
	300 MHz	450 MHz
chip-set	Intel 440 BX	Intel 440 GX
DRAM	8 GByte	48 GByte
network	extended PCI/SCI LC2 interface cards	the same
	500 MByte/s uni-directional	the same
	4×8 torus	8×12 torus
peak perf.	19.2 GFlops	86.4 GFlops

Table 21.1. PSC-64 and PSC-192 hardware configuration

processors are clock-locked by a multiplier of 4.5, resulting in a system bus frequency of 66 MHz.

Before deciding on the system components, we have evaluated about half a dozen Intel chip-sets for their utilization of the PCI bus [16]. In our tests, the Intel 440 BX and 440 GX chip-sets gave best results on the 32-bit PCI/SCI adapter cards with 33 MHz. Other chip-sets, like Intel 440 LX, for example, do not adequately support the PCI bus. Also the Orion chip-set, which was commonly used in the Intel quad-board (4-way processor) systems, provided very poor throughput (i.e. less than 25 MByte/s). This prompted us to give up our initial plans to build a cluster with quad-board systems. The quad-board PCs are now employed as front-end systems for the cluster. They are not connected to the SCI ringlets, but act as gateways between the PSC's Fast Ethernet and our institute's backbone network.

The PSC-192 system, which was installed one year later, has a similar architecture but with improved hardware components (Table 21.1). It consists of 96 Siemens Primergy servers, each equipped with two 450 MHz Intel Pentium II with 440 GX chip-set, 512 MByte of main memory and a local hard disk drive. With its 100 MHz bus frequency and the increased CPU clock rate of 450 MHz, a PSC-192 node is in practice about 50% faster than a PSC-64 node.

21.2.2 SCI Interconnect

The PSC Servers contain 4th generation PCI/SCI adapter cards (D308 revision D) that have been designed by Scali Computer AS for large clusters. They are based on Dolphin's CluStar PCI technology (PCI/SCI card D310), but have an additional connector for card extensions to a so-called *Mezzanine* board with an additional link controller. The two link controllers and the PCI/SCI bridge (PSB) are connected via an internal B-link bus. Thus,

the two link controllers of a node can be used to build a 2D torus, where each node is connected to an x- and y-ringlet, giving a distributed switch.

With a maximal cable length of two meters (parallel copper), the SCI link speed is set to 500 MByte/s. For reliable communication over longer distances, we have also experimented with a reduced link speed of 400 MByte/s.

21.2.3 Software Configuration

The PSC Servers are operated in multi-user mode under Solaris, Linux or Windows NT. Multi-user mode is implemented by logically linking compute nodes to the user's partitions. In principle, it is even possible to run different operating systems on different partitions at the same time. With this operating mode, all resources local to a node are managed by the local operating system. For the benchmark results presented here, we have used Solaris X86.

operating systems	Solaris X86, Linux, Windows NT
message passing libraries	ScaMPI, Active Messages, PVM
shared memory libraries	Yasmin
compilers	pgcc, pgCC, pgf77, gcc, g++, g77
math libraries	ScaLAPACK
program development	TotalView, Vampir, SProf
resource management	CCS

Table 21.2. PSC software configuration

Table 21.2 shows the PSC software configuration. The message passing environments MPI and PVM come in two versions, one taking full advantage of the SCI interconnect (ScaMPI, see Chapter 14) and the other running on Fast Ethernet (MPICH). For shared memory applications, the *Yasmin* environment [4] provides a programming interface for shared memory regions, semaphores and other synchronization mechanisms.

Both clusters are operated under the resource management system *CCS* (Chapter 26) which provides system administration facilities and user-friendly system access with a variety of schedulers for interactive, batch, or mixed operation modes.

For the fine-tuning of application codes we have implemented the *SProf* toolset [15] which helps to determine performance bottlenecks, especially in the memory hierarchy (L1 and L2 caches) of large parallel systems.

21.3 Standard Benchmarks

21.3.1 Low-Level MPI Benchmarks

Low-level benchmarks are used to characterize the performance of basic building blocks (processor, memory, disk, network), computing nodes or complete HPC systems. In a first test series we have run an MPI ping-pong benchmark to measure the communication latency and bandwidth between arbitrary pairs of nodes. As usual, we define latency as half of the time needed to transfer a zero-length message between a sender and a receiver back and forth. The effective bandwidth is determined with the same ping-pong test, but using various message sizes.

Two communication modes have been measured, a uni-directional data transfer that utilizes only one communication direction of a link at a time and a bi-directional communication with simultaneous data transfers in both directions.

Communication Latency. The latency histogram in Figure 21.3 shows an average latency of $18.7\mu s$ with some small variance due to sporadic activities of the nodes' operating systems. Interestingly, on an unloaded network we found the latency to be almost independent of the positions of the communication partners. No significant time delay was observed when changing from an x- to a y-ringlet or vice versa.

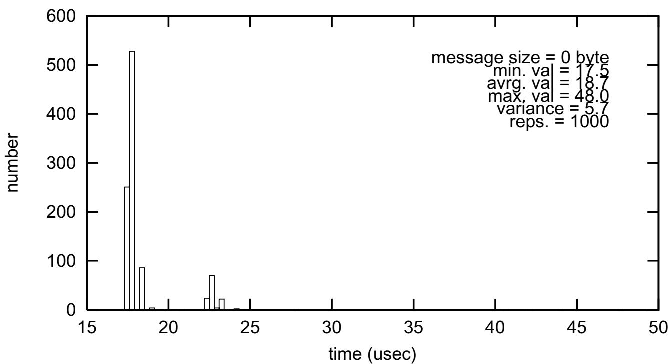


Fig. 21.3. Communication latency of ScaMPI on PSC-64

Using MPICH on the same nodes but with Fast Ethernet gave a ten to twenty times higher latency (250 to $400\mu s$). This must be attributed to the slower communication speed on the shared medium, to the TCP/IP protocol stack, and to the non-optimal, but portable MPICH implementation.

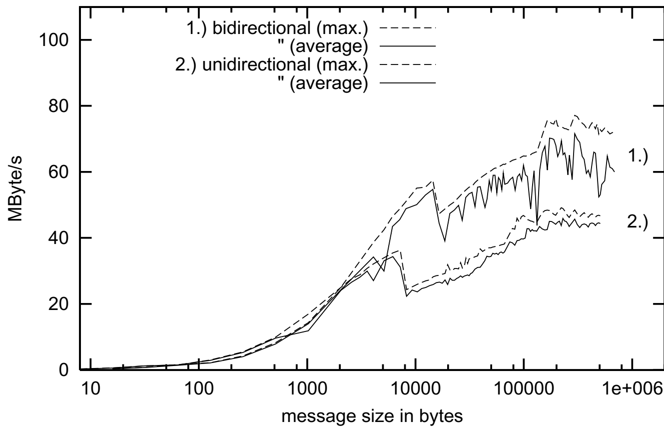


Fig. 21.4. Point-to-point ScaMPI communication on PSC-64

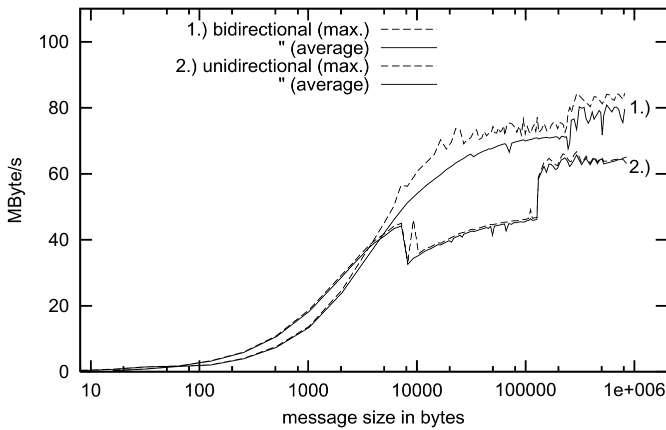


Fig. 21.5. Point-to-point ScaMPI communication on PSC-192

Communication Bandwidth. Figures 21.4 and 21.5 show the best and average communication bandwidth of several thousand test runs on PSC-64 and PSC-192. Caching effects and the three different ScaMPI transport protocols for various message sizes [13] gave the characteristic curves. Notice that in the uni-directional case, the maximum and average performance curves are closer together. With bi-directional communication, we spotted many more test instances with bad performance (possibly due to message transfer conflicts on the SCI ringlets). As before, the 45 MByte/s uni-directional and 65 MByte/s bi-directional bandwidth with ScaMPI on PSC-64 is an order of magnitude higher than that of MPICH on Fast Ethernet (5 to 7.5 MByte/s). On PSC-192, we have measured 64 and 82 MByte/s, respectively, both with a marginally lower communication latency.

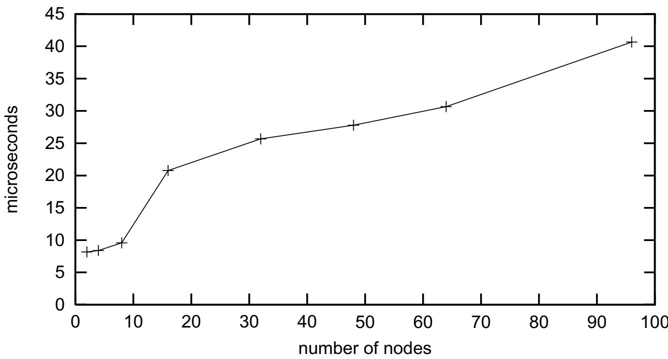


Fig. 21.6. Barrier synchronization time on PSC-192

Barrier Synchronization. Figure 21.6 shows the performance of MPI barrier synchronization on various groups of up to 96 nodes. All data points have been averaged over several thousand trials because there is a high variance in the measured synchronization times. In practice, the time required for one single barrier cannot be predicted, especially when a large number of nodes is involved. This is because of spontaneous activities of the nodes' operating systems.

21.3.2 Parallel Linpack

The parallel Linpack benchmark [5, 6] is commonly used as a yardstick to measure the performance of the world's fastest supercomputers. The task is to solve a dense system of linear equations as fast as possible. In contrast to many other benchmarks, the implementor may choose any suitable algorithm and problem size. New Linpack results are published twice a year [7].

We have implemented a variant of the standard Gaussian elimination with partial pivoting. The $n \times n$ matrix is mapped onto the mesh of nodes with the classical data distribution scheme, generally referred to as a two dimensional block-wrapped, or block-cyclic, matrix decomposition. The operations on the $b \times b$ sub-matrices provide a good system utilization, including the complete memory hierarchy in the nodes. Moreover, the data distribution gives an almost perfect static work-load balance over the nodes. The load balancing between the two processors of a node is done by multi-threading. Two computation threads for the elimination task, one thread per processor, are controlled by an intelligent scheduling instance to keep them busy as long as possible. A couple of other threads are used to handle communication and update processes. For more details on our algorithm, see [17].

The matrix-matrix multiplication in the solution of the dense system of equations can be performed at 320 MFlop/s when using both CPUs of a node. Due to an implicit synchronization of the nodes during the search of

no. of CPUs	mesh size	Overall LU perf. (MFlop/s)	Overall Linpack perf. (MFlop/s)	Linpack perf. per node (MFlop/s)	matrix dimension (n)
64	8 × 4	7859	7811	244	27392
48	6 × 4	5685	5658	236	23808
32	4 × 4	4009	4003	250	19328
16	4 × 2	2042	2037	255	13952

Table 21.3. Parallel Linpack performance on PSC-64

a pivoting element in the rows of the matrix, the parallel Linpack program looses about 10% in efficiency. As can be seen in table 21.3, the efficiency drops by 4% when increasing the number of processors by a factor of 4.

21.3.3 FFT Benchmarks

FFTW is a C subroutine library [8] for computing the Discrete Fourier Transformation (DFT) in one or more dimensions—of both real and complex data—and of arbitrary input size. The performance of FFTW is reported in terms of "MFLOPS" for various transform sizes n, see Figure 21.7. Note that this MFLOPS rate is not directly comparable to the commonly used MFlop/s performance. It is rather an abstract performance measure, defined as $5 \log_2 N/(time\ of\ an\ FFT)$.

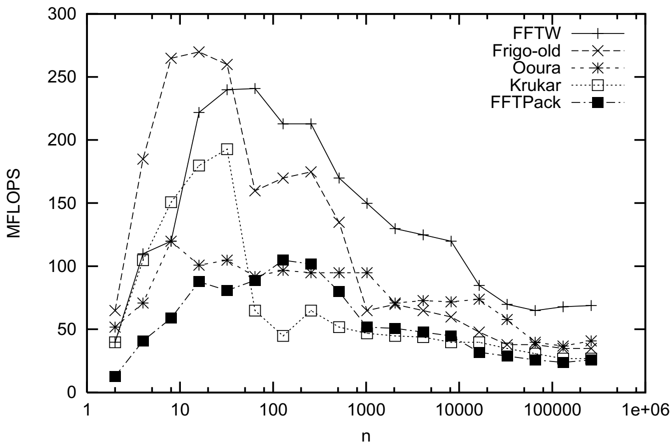


Fig. 21.7. FFT on a single PSC-64 node

The FFTW library is available for single processor systems, shared memory-, and distributed memory systems. We have chosen the MPI va-

riant. Figure 21.8 shows the speedup of the transposed FFT of a 3D matrix on 1 to 32 nodes, that is, on 2 to 64 processors. With ScaMPI, we measured an efficiency of 60% with a problem size of only 240^3 on 32 nodes. While larger problem sizes yield better results, we have chosen this small size because it fits into the memory of a single node.

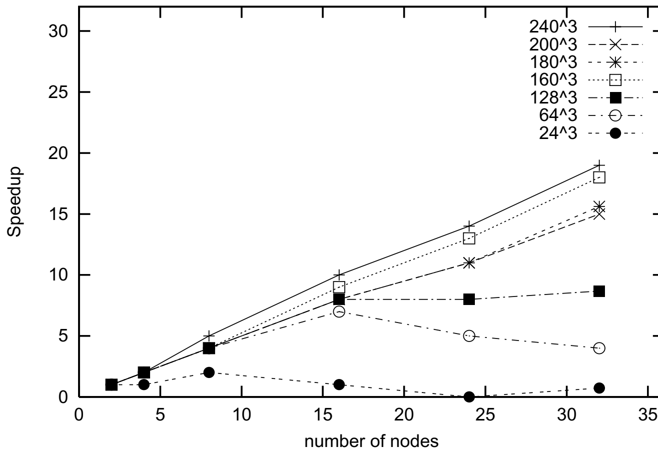


Fig. 21.8. FFTW on PSC-64 for various matrix sizes

21.3.4 HINT Benchmark

HINT or Hierarchical INTegration is a benchmark tool from Ames Laboratory [9] to gauge the overall performance of a variety of computers. The HINT benchmark fixes neither the problem size nor the execution time of the problem to be solved. It measures the performance of a computer across all memory levels (caches, main memory, swap space, etc.). The benchmark uses interval subdivision to find upper and lower bounds of the function $f(x) = (1 - x)/(1 + x)$ using only the monotone decreasing property of the function. Quality is defined as the reciprocal of the difference between the upper and lower bounds. At each time step the quality improvement is calculated with respect to the time spent for the calculation. The output of the benchmark is a graph of QUality Improvements Per Seconds (QUIPS) over the memory used for the calculation. NetQUIPS summarizes the QUIPS over time.

Figure 21.9.a shows the HINT performance plot for different data sizes and types on a single processor. With 32-bit integer we measured a performance value of 14.38 MQUIPS and with 64-bit floating point it has 10.74 MQUIPS. Note that these performance values are single numbers that include

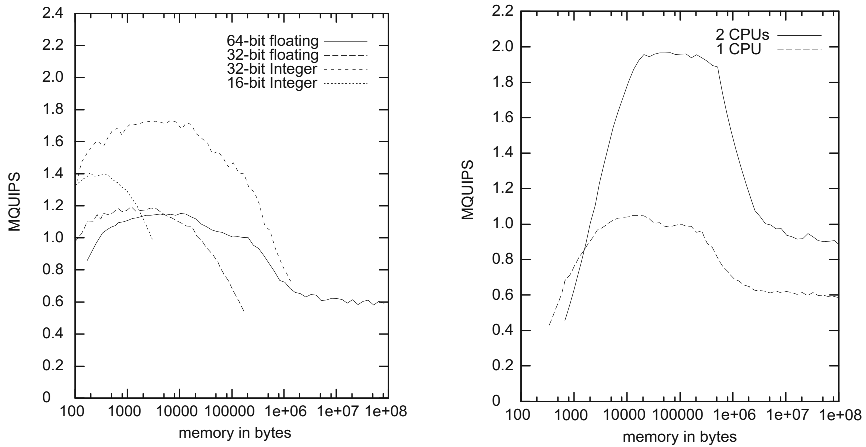


Fig. 21.9. a.) Sequential HINT performance on one CPU for different data types b.) MPI implementation of HINT on a single node using one or two CPUs with double precision floating-point data

the system performance of the whole memory hierarchy, i.e. the performance values (measured in MQUIPS) are given by the integral of the curves.

On parallel systems, the HINT benchmark uses domain decomposition to distribute the function domain. Each processor calculates a scattered portion of the domain. Figure 21.9.b shows the parallel HINT on a node with one (9.75 NetMQUIPS) or two processors (15.54 NetQUIPS) for double precision floating-point operations[1].

21.4 Applications

Complementing the described low-level benchmarks, we have run a number of practical applications on the PSC Servers. In this section, we report on the performance of a preconditioned CG solver with domain decomposition, an application for protein structure comparison and a parallel chess program.

Numerical Flow Simulation. In numerical flow simulation there are many methods for solving the Navier-Stokes equation. Splitting schemes result in linear and non-linear systems of equations. We examined the linear case for solving the pressure correction step, which is described by a mixed boundary value Poisson problem [2]

$$\begin{aligned} -\Delta u(x,y) &= f(x,y) \text{ in } \Omega \\ u(x,y) &= 0 \text{ on } \partial\Omega \end{aligned} \tag{21.1}$$

[1] For comparison, HINT results with other systems can be found on http://www.scali.com/html/performance.html

Classical methods for solving this kind of equations, like Conjugate Gradient (CG), Jacobi, or Gauss-Seidel solvers are inefficient, because the number of iterations depends on the space step-size of the discretized domain. Therefore preconditioned conjugate gradient methods or multigrid methods are used. Both have their specific advantages. In the case of adaptive mesh refinement we use a preconditioned conjugate gradient method with domain decomposition (DD-PCG).

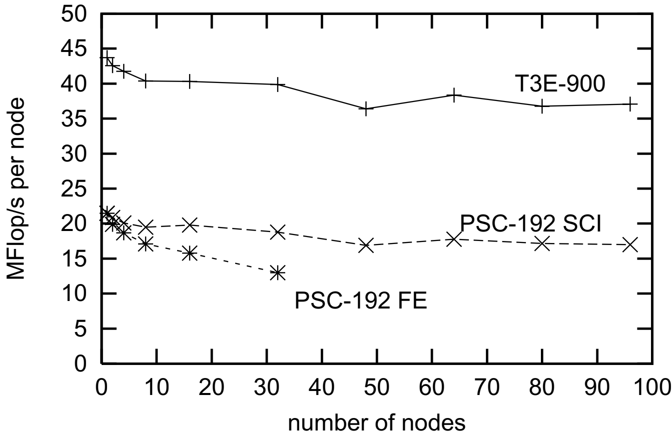


Fig. 21.10. Performance comparison of a DD-PCG solver on a Cray T3E-900 and the PSC-192 with SCI and Fast Ethernet (FE)

The graphs in Figure 21.10 show the scalability of the conjugate gradient boundary iteration method [2, 3] solving equation 21.1 on a fixed domain $\Omega = [0, 2] \times [0, 1]$. The same DD-PCG solver code has been run on

– PSC-192 with ScaMPI on SCI
– PSC-192 with MPICH on Fast Ethernet (up to 32 nodes)
– SGI/Cray T3E-900 with Cray MPI running on 450 MHz DEC Alpha 21164 processors and 128 MByte main memory per node

Figure 21.10 shows the scaled application performance, i.e. the application input is scaled up with the nodes and the available user memory. The same DD-PCG solver ($\approx$ 70 MByte user memory per node) has been run on the T3E and the PSC-192. The performance is given in MFlop/s per processor relative to the number of operations of the sequential algorithm. Hence the data points also show the scalability of this specific implementation.

While both systems have the same nominal peak performance of 900 MFlop/s per node, their sustained performance differs quite substantially. This is attributed to the optimized memory interface of the T3E for strea-

nodes	SCI	Fast Ethernet
2	1.5	1.4
4	2.6	2.4
8	4.7	3.4
16	9.0	7.7

Table 21.4. Speedup results for MSAP on PSC-64

med memory accesses. The PC components used in the PSC-192, in contrast, have a standard memory hierarchy (L1-, L2-cache, main memory) with no hardware support for consecutive memory fetches from main memory. Hence, we found it more efficient to use only one PSC processor for the computation, while the second processor is used exclusively for communication. When using both processors in a node, the memory bandwidth becomes a bottleneck in this kind of application.

More important than the absolute node performance is the scalability of the DD-PCG solver on the two systems. Note that both curves are of almost flat, indicating an almost perfect scalability for up to 96 nodes! The communication latency (16 μs over MPI) is nearly the same on both systems, whereas the communication bandwidth of the T3E is about twice as high as that of SCI. Moreover, each T3E node may communicate simultaneously in all three directions of the 3D torus, while a PSC node can only serve one communication ring at a time.

The lower graph shows the MPICH version running on Fast Ethernet. Clearly, the application does not scale due to insufficient communication performance.

Protein Structure Comparison. As another example for a practical application, we have run *MSAP*, a parallel variant of *SAP* [18] which is used in rational drug design for the comparison of tertiary structures of proteins. MSAP identifies the similarity of a given protein model—which may have been obtained by computer simulations or by experiments—to all known protein structures in a database. When some similarity has been found, this usually gives information on the function of the unknown protein, as the function of almost all proteins with experimentally determined structure is known.

MSAP is implemented in C++/MPI and has been compiled for ScaMPI on SCI and MPICH on Fast Ethernet. Both variants have been run on PSC-64 with up to 16 nodes. Up to four nodes, both variants scale nearly identical (Table 21.4). With 16 nodes, the relative speedup on SCI is 9.0, while the variant running on Fast Ethernet provides a 15% lower speedup.

Computer Chess. Game playing programs are often seen as excellent benchmarks for the efficiency of parallel systems, because they are

- CPU bound, because the score of each node in the look-ahead tree must be assessed with a complex evaluation function,

– memory bound, because it is faster to store previously evaluated nodes in a large transposition tables than to re-evaluate them when they are needed again,
– communication bound, because the transposition table is distributed over the parallel system and table accesses must be fast enough to vindicate the effort.

Chess programs build a large decision tree when exploring all possible moves and their consequences. Even with modern pruning techniques, the tree size grows exponentially with the exploration depth so that only a section of the game tree can be searched.

In our benchmark suite, we have used the chess program *ConNerS* [11] which employs a non-classical technique for incremental strategic search, called “Controlled Conspiracy Number Search”. Depending on the number of processors, ConNerS plays at a Grandmaster level. On a small (ten processor) SUN UltraSPARC II with 300 MHz, for example, ConNerS has a rating of approximately 2580 Elo points.

Again, we compared a ScaMPI and an MPICH variant on PSC-64 . With 32 processors, the SCI variant of ConNerS gave a 16.7-fold speedup while Fast Ethernet provides a factor of only 9.8. This is attributed to the many small messages sent by ConNerS, where a small communication latency is of prime importance.

For another cross-check, we have run ConNerS on a Parsytec CC-48 with 40 PowerPC 604 processors clocked at 133 MHz. Each processor is connected to a 1 GBit/s HS-link with a fat mesh of Clos topology. On this system, we were also able to achieve a 50% efficiency on 40 processors, but with a much lower overall performance due to the inferior PowerPC 604 processors. As a result, the whole Parsytec CC-48 with 40 processors is only four times as fast as a SUN UltraSPARC II with 300 MHz. Each Intel Pentium II in the PSC-64, in contrast, as about the same performance as an UltraSPARC.

21.5 Summary

In this chapter, we have presented first empirical results on the operation of two high-performance compute servers with standard Intel processors and SCI interconnect, one with 32 off-the-shelf PC-boxes and the other with 96 motherboards with server components. It seems that the use of common off-the-shelf technology (COTS) is a very viable alternative to dedicated hardware—even in the field of high-performance parallel computing. Both, the computing and the communication performance of the PSC Servers is comparable to that of dedicated HPC systems, but at a much lower cost.

Even more important, a multitude of well-engineered and practice-proven software packages is available for COTS. As an example, we run three popular operating systems (Linux, Solaris, and Windows NT) on our PSC Servers.

However, the use of full-fledged operating systems on the single nodes has also disadvantages too: The operating system takes much memory space on each node and, more important, undesirable side-effects (UNIX watchdog, automounter, etc) may slow down the overall system performance, especially when performing group communication.

Also, some difficulties remain with the resource management of such clusters. Currently, users have the alternative between standard cluster management software (e.g. LSF, Codine, Condor) and our Computing Center Software CCS (Chapter 26). Aiming primarily at high-throughput computing on loosely coupled clusters, the cluster management software does not fully exploit the capabilities of dedicated SCI servers, while CCS is still a university product without commercial support.

Acknowledgments

Special thanks to Stefan Blazy, Jan Hungershöfer, Max Ibel (UBC), Ulf Lorenz, and Jens-Michael Wierum for providing valuable information on the applications and for supporting our work so generously. Also thanks to Scali Computer for providing us with the latest version of their ScaMPI software.

References

1. D. Bailey, J. Barton, T. Lasinsk, and H. Simon. *The NAS Parallel Benchmarks.* Technical Report RNR-91-002 Revision 2, 1991.
2. S. Blazy, W. Borchers, and U. Dralle. Parallelization methods for a characteristic's pressure correction scheme. In: H. Hirschel (ed.): *Flow Simulation with High-Performance Computers II. Notes on Numerical Fluid Mechanics*, vol. 38, Braunschweig, Germany, Vieweg-Verlag, 1996.
3. J.H. Bramble, J.E. Pasciak, and A.H. Schatz. The construction of preconditioners for elliptic problems by substructuring I., *Math. Comput.* 47, pp. 103-134, 1986.
4. R. Butenuth and H.-U. Heiß. Shared memory programming on PC-based SCI clusters. *Procs. of the SCI Europe'98*, Bordeaux, France, 28.-30. Sept. 1998.
5. J.J. Dongarra. The LINPACK benchmark: An explanation. *Evaluation in Supercomputers*, Chapman and Hall, 1990, pp. 1–21.
6. J.J. Dongarra. *Performance of Various Computers Using Standard Linear Equations Software.* Technical Report CS-89-85, University of Tennessee, http://www.netlib.org/benchmark/performance.ps.
7. J.J. Dongarra, H.W. Meuer, and E. Strohmaier. *Top500 Supercomputer Sites.* http://www.netlib.org/benchmark/top500.html.
8. M. Frigeo, and S.G. Johnson. FFTW: An adaptive software architecture for the FFT. http://theory.lcs.mit.edu/~fftw/, *ICASSP 98*, 1998.
9. J.L. Gustafson and Q.O. Snell. *HINT: A New Way To Measure Computer Performance.* Ames Laboratory, http://www.scl.ameslab.gov/Personnel/john.html, 1995.
10. O. Heinz and J. Simon. SCI multiprocessor PC cluster in a WindowsNT environment. *Supercomputer Journal*, (68, XIII-2), 1997.

11. U. Lorenz and V. Rottmann. Parallel Controlled Conspiracy Number Search. *Advances in Computer Chess 8*, pp. 135–147, 1997.
12. NASA Ames Research Center: *The NAS Parallel Benchmarks.* http://science.nas.nasa.gov/Software/NPB/
13. Scali Computer: *ScaMPI User's Guide.* 1997.
14. Scali Computer: *SCALI Affordable Supercomputing.* http://www.scali.com/
15. J. Simon, R. Weicker, and M. Vieth. Workload Analysis of Computation Intensive Tasks: Case Study on SPEC CPU95 Benchmarks. *Euro-Par'97*, Springer LNCS 1300, 1997, pp. 971–984.
16. J. Simon and O. Heinz. Experiences with a SCI Multiprocessor Workstation Cluster. *ARCS'97*, VDE-Verlag 1997, pp. 189–199.
17. J. Simon and J.-M. Wierum. Sequential Performance versus Scalability: Optimizing Parallel LU-Decomposition. *HPCN'96*, Springer LNCS 1067, 1996, pp. 627–632.
18. W.R. Taylor, T.P. Flores, and C.A. Orengo. Multiple Protein Structure Alignment. *Protein Science*, vol. 3, 1994, pp. 1858–1870.

22. Shared Memory Parallelization of the GROMOS96 Molecular Dynamics Code

Marcus Dormanns

RWTH Aachen, Lehrstuhl für Betriebssysteme, Kopernikusstr. 16,
D-52056 Aachen, Germany
email: contact@lfbs.rwth-aachen.de
http://www.lfbs.rwth-aachen.de/

22.1 Introduction

This chapter describes the parallelization of GROMOS96, a molecular dynamics simulation code, on SCI-interconnected cluster architectures.

The simulation of the dynamics of molecular systems is one of the central topics in scientific high-performance computing. It has been subject to parallel processing already for a long time. GROMOS96 [16] is a well-known code in this area with a quite long history. It is the successor of GROMOS87, which has been re-designed for improved functionality and performance, including a parallel implementation for shared memory Silicon Graphics multiprocessors [7].

Due to its irregular and time-varying data structures that are processed by different algorithms, this code is well suited to discuss several aspects of application parallelization on the considered cluster platform. The work presented here is part of a larger application parallelization project [13] that is concerned with codes from different areas, e.g. a module of a decision support system [4], an acoustics simulation code [15] and earthquake simulation [14].

The parallelization of GROMOS96 exploits the shared memory capabilities of the SCI cluster platform via the *Shared Memory Interface* (*SMI*; see [5] and Chapter 16) parallelization library. Developing techniques for NUMA shared memory parallelization is an interesting subject because most of the recent small to medium size parallel systems provide a (logical) shared address space (e.g. SGI Origin, HP Exemplar, Data General AViiON NUMALiiNE, Sun Enterprise server and Sequent NUMA-Q). Today, there exist only few purely distributed systems (e.g. IBM SP2) which mainly target at the very high-end market and therefore represent a different class of machines than cluster systems.

Section 22.2 provides some more information about GROMOS96 and its code structure. In Section 22.3, two different aspects of the GROMOS96 parallelization are described: The first one concerns the overall principles and software engineering issues, while the second one sketches the parallelization of the kernel algorithms. Section 22.4 then discusses the resulting performance

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 383-396, 1999
© Springer-Verlag Berlin Heidelberg 1999

and compares it to similar parallelization efforts. Finally, some conclusions are drawn in Section 22.5.

22.2 The GROMOS Code

22.2.1 General Code Characteristics

The purpose of molecular dynamics simulation is to track the dynamics of a molecular ensemble over time whose individual atoms interact via several types of forces. Depending on the concrete application, results of interest are the particles' trajectories (or time-evolving quantities that can be computed from them) as well as the final configuration.

The whole software package consists of several programs. Besides pre- and post-processing programs, the main component is the actual molecular dynamics simulation program. This alone contains 31 source code modules with about 42,000 lines of Fortran 77 code altogether. The difference to the former GROMOS87 code becomes obvious, when comparing these quantities to the 22 modules with just about 9,000 lines of Fortran 77 code of GROMOS87.

GROMOS87 was already subject to several parallelization efforts. UH-GROMOS and EulerGromos have been developed at the Texas Center for Advanced Molecular Computation (Univ. of Houston) [3, 10, 12]. Furthermore, GROMOS87 has been parallelized within the framework of the EU-funded *Europort* project [6, 11]. Both efforts led to message passing programs. The new GROMOS96 code has been parallelized only once, using threads for shared memory Silicon Graphics multiprocessors [7].

22.2.2 Structure of the Code

The code structure is highly pre-determined by the underlying physics and the numerical solution method (see e.g. [1, 16]). Denoting the spatial position of an atom i with mass m_i at a specific point in time t by $r_i(t)$, a set of coupled nonlinear differential equations is solved in the time domain that determine the atoms' trajectories due to Newton's equations of motion considering an interaction potential E:

$$\frac{\partial^2}{\partial t^2} r_i(t) = -\frac{\nabla E(t)}{m_i} \qquad i = 1, 2, 3, \ldots \tag{22.1}$$

This is done in a time-step fashion. Starting from a given initial configuration, for each point in the discretized time the forces on each particle are computed and accumulated. From these, Newton's equations of motion allow to determine position and velocity of all particles at the following time step by integration.

Forces can be divided into two types:

Module	count	time
main		
load configuration data	1	~0
perform T time-steps of simulation		
all t_{pl} time-steps: assemble pair-list and compute long-range interactions		
solute-solute and -solvent	T/t_{pl}	13.8
solvent-solvent	T/t_{pl}	33.3
compute short-range interactions		
solute-solute and -solvent	T	10.4
solvent-solvent	T	36.1
SHAKE	4+2T	2.6
integrate positions to next time-step	T	~0
all t_{out} time-steps: write data to file	T/t_{out}	~0
write final configuration data to file	1	~0

Fig. 22.1. GROMOS96 code structure and the modules' computational complexities (in % of the total time) and frequency count for a *thrombin* molecule (3,078 atoms) with 5,427 solvent water molecules.

– Non-bonded interactions that act between any pair of atoms. These can be further divided into long-range and short-range interactions, depending on their decay rate with increasing distance.
– Bonded interactions, resulting from chemical bonds between atoms. They capture e.g. bond lengths, bond angles and bond dihedral angles.

Short-range interactions are typically neglected for all atom pairs beyond a certain cut-off radius. All relevant atom pairs are kept in a so-called *pair-list* to allow an efficient evaluation of the forces. Due to the smoothness of the dynamics, it is sufficient to update the pair-list only every t_{pl} time steps (e.g. $t_{pl} = 10$). Analogously, long-range interactions are evaluated only from time to time and assumed to be constant in between. In GROMOS96, long-range interaction evaluation and pair-list construction is performed within the same procedure.

Additional to these mechanisms, it is often required to restrain some degrees of freedom of the molecular ensemble, e.g. bond lengths, that would undergo forbidden modifications within the simulation process. This is achieved with an iterative procedure, commonly called *SHAKE*. SHAKE adjusts the ensemble iteratively according to the restrictions. The code structure is summarized in Figure 22.1 for a molecular ensemble with the protein molecule *thrombin* (3,078 atoms) with 5,427 solvent water molecules (altogether 19,359 atoms). Figure 22.1 also shows the number of calls of and the time spend in the modules.

22.3 Parallelization

The parallelization of GROMOS96 is based on the *Shared Memory Interface* (*SMI*) library introduced in Chapter 16. It was done in accordance to the step-by-step parallelization strategy implied by SMI. This methodology has several advantages:

– Reduced complexity. Especially for the parallelization of given sequential codes (in distinction to the development of a parallel application from scratch) it is critical to deal with all data structures and sub-algorithms at once.
– Scalability of the parallelization process. The performance of the parallel code should scale with the amount of work spent on the parallelization process. A detailed analysis of the time spent in the sequential code parts may result in huge performance improvements by parallelizing the critical code sections only. This is especially desirable in projects with limited (financial) resources.

Shared memory is an essential requirement for both issues. Dealing with a message passing parallelization that comes along with partitioned and distributed data structures, it would be impossible to proceed in such a step-by-step process. The possibility to start with the parallelization of just a few code parts is an enormous advantage especially for a code like GROMOS96, which consists of a couple of different algorithms. Although contributing not considerably to the computational complexity, some of them are quite complex to parallelize and can be omitted in the first steps. Therefore, SCI-based NUMA shared memory clusters show much more advantages than just performance considerations would suggest.

In the following, the individual steps undertaken in the GROMOS96 parallelization effort are described.

22.3.1 Starting with Parallelism and Coordinating I/O

The parallel execution environment is set up by initializing SMI and requesting several parameters from it, e.g. the total number of processes, the process ranks, etc.

GROMOS96 uses several streams of output. Some are written into certain files (e.g. trajectory data of the atoms over time), one goes to the console (simulation parameters, error messages, ensemble-averaged quantities for individual time-steps as well as averaged over the entire simulation run). For parallelization related output like performance numbers, debug output, etc., the standard error stream is used. Using SMI's capabilities, standard error is re-directed to the window-based front-end (see Chapter 16). Standard output is re-directed to files, ensuring a different file name for each process automatically. File output is performed by a single process only. This is possible

since within the shared data programming model, each process has access to all data. Clearly, this approach is not scalable in terms of Amdahl's law. But the major focus of this parallelization effort was to study the algorithm parallelization.

22.3.2 Parallelization of the Interaction Calculation Kernels

The two most time consuming modules are:

– pair-list construction (including long-range interaction evaluation) and
– short-range interaction evaluation.

The pair-list is constructed not on the basis of individual atoms but on the basis of small clusters of atoms that together possess a nearly neutral charge, the so-called *charge-groups*. A charge-group is, for example, an entirely solvent water molecule. Besides numerical advantages, this also reduces the problem size for pair-list construction by a factor of about three.

The pair-list construction that is coupled with long-range force evaluation (if this feature is requested) is a simple $O(N^2)$ algorithm that tests the distance of all pairs of charge-groups. If it is small enough, a long-range force contribution is evaluated and accumulated to a long-range force array for the considered atoms. If it is even smaller than the short-range interaction cut-off radius, the respective charge-group pair is added to the pair list.

The short-range interaction evaluation module consists of a loop over all entries in the pair-list. For each charge-group pair, all interactions between all constituting atoms are evaluated and accumulated to the short-range force array.

Three properties of GROMOS96 are worth noting, because they influence the parallelization strategy:

– Due to Newton's law of *actio and reactio*, forces between atoms are antisymmetric, i.e. identical in magnitude but contrary in direction. Once a force has been evaluated, it is crucial to accumulate it to both atoms under consideration to gain performance. Therefore, it is impossible to parallelize the application in a way that no two processes perform accumulations to the same entries in the force arrays.
– The GROMOS96 implementation relies on a specific mapping of atoms to indices in the force arrays. One major assumption is that all solute atoms precede the solvent atoms. Furthermore, all atoms of each individual solute molecule (if there are several) are consecutive in the force array, again in a specific common order. Departing from this would result in major implementation changes although it might be very reasonable considering parallelization solely.

Construction of the pair-list and evaluation of long-range forces. The major data structures affected by a parallelization of this module are:

– the array of all atoms' forces that is subject to accumulation operations and
– the pair-list.

The basic principle is to split the outer-most loop of the $O(N^2)$-algorithm among the processes. This means that several processes may contribute to an atom's long-range force. To enable this, the long-range force array is placed in a globally accessible SCI shared memory region. The region is allocated with the `BLOCKED` directive of SMI, i.e. it is assembled from equally-sized segments on each machine to balance the number of remote memory accesses among the processes. Then, an essential requirement for correctness is to perform accumulations from different processes to identical atoms under mutual exclusion.

The straight-forward way to ensure this is to guard each accumulation operation by a mutex. Although SCI allows to implement efficient synchronization mechanisms, their cost (on the order of 50 μs) is much to high to allow the guarding of each single accumulation operation. We therefore blocked the execution of the loop (see e.g. [9]). The resulting temporal locality is advantageous in several ways:

– The usage of the cache is improved.
– For each atom that is processed within this algorithm, quite an amount of data that influences the computation has to be looked up in several different data structures. For a small and deterministic number of atoms it is possible to do this just once during the processing of the entire loop block and to keep it in a suitable data structure during that phase. This is called a *software cache*.
– If some kind of software-caching is done, it can be expanded to also capture the atoms' forces. An accumulation to the global force array is then performed for all atoms at the end of the processing of an entire loop-block, capturing all partial forces at once that result from all interblock interactions. This compensates for the necessary synchronization operation and also for the overhead due to remote memory accesses.

The scheduling of loop blocks to processes is performed by SMI's loop-scheduling facilities that minimize load imbalance while introducing only a minimum of scheduling overhead. The resulting code fragment is sketched in Figure 22.2.

Besides long-range forces, the results are per-process pair-lists corresponding to the considered atom pairs within each process. This implicit partitioning of the overall pair-list is already the basis for the parallelization of the short-range interactions. So far, load balance has been optimized for pair-list computation itself, but not for the resulting pair-lists. This will eventually result in load imbalance during the short-range force computation phase.

```
SMI_Switch_to_sharing(LongRangeForceArray)
SMI_Loop_schedule_init(1, NChargeGroups);
While work Do
   SMI_Get_iterations(&OblockIdxMin, &OblockIdxMax);
   Load data of atoms OblockIdxMin,...,OblockIdxMax
   For Iblock=1 To Nblocks Do
      IblockIdxMin = Iblock * BlockSz
      IblockIdxMax = min{NChargeGroups, (Iblock+1)*BlockSz-1}
      Load data of atoms IblockIdxMin,...,IblockIdxMax
      For i=OblockIdxMin  To OblockIdxMax  Do
         For j=IblockIdxMin  To IblockIdxMax  Do
            Process atom-pair (i,j):
            - compute distance
            - compute long-range force if distance small enough
            - insert into pair-list if distance small enough
         End For
      End For
      SMI_Mutex_lock(...)
         Accumulate partial forces of atoms IblockIdxMin,...,IblockIdxMax
      SMI_Mutex_unlock(...)
   End For
   SMI_Mutex_lock(...)
      Accumulate partial forces of atoms OblockIdxMin,...,OblockIdxMax
   SMI_Mutex_unlock(...)
End While
SMI_Switch_to_replication(LongRangeForceArray)
```

Fig. 22.2. Pseudo-code of the parallelized pair-list construction and long-range force evaluation algorithm.

To hide the parallelization of this module from the rest of the application, SMI's capabilities to switch between sharing and replication of memory regions are used. Although not mentioned in particular, this parallelization scheme is applied to solute-solute and solute-solvent as well as pure solvent-solvent interactions in the same way.

Evaluation of short-range forces. The work partitioning of the short-range force evaluation module is already pre-determined by the existence of a partial pair-list in each process. Analogously to the pair-list construction, the central data structure that is processed in parallel is the short-range force array for all the atoms. So, the parallelization of this module is faced with similar problems:

– Minimize the number of force array accesses because a fraction of them are expensive accesses to remote memory.
– Minimize the number of lock/unlock operations that are necessary to guarantee atomicity of accumulation operations from different processes.

Blocking, as it was done in the pair-list construction module, is not straight-forward, because the pair-list represents an irregular grid structure (considering each charge-group as a node and drawing an edge between all nodes with an Euclidean distance smaller than the cut-off radius). The ori-

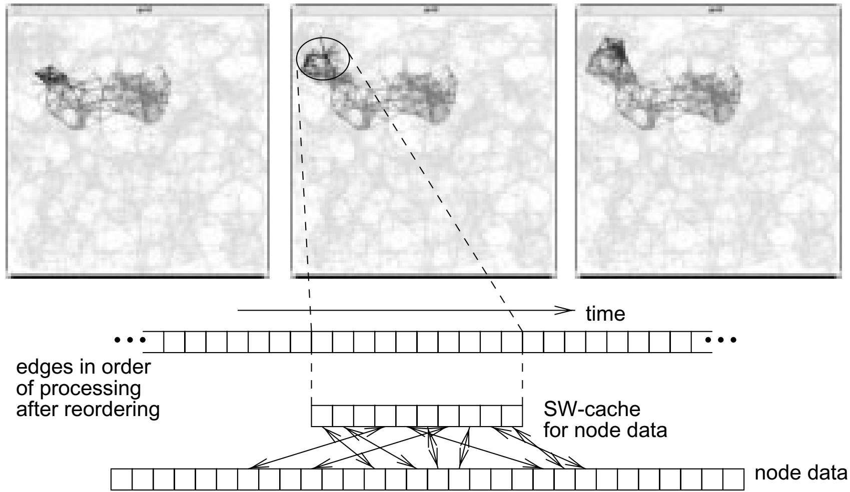


Fig. 22.3. Three consecutive snapshots of temporal locality optimized grid traversing. Black edges denote the latest visited edges (with a cache-size of 64 nodes).

ginal sequential code used a node-oriented procedure that processed interactions in the order of the first node. Denoting the number of charge-groups by G and the pair-list length by P, this requires $G + P$ load operation of atom data with a proportional amount of cache misses in the sequential case and additionally a proportional amount of remote memory accesses in the parallelized case. P is on the order of one magnitude higher than G, i.e. each node possesses some tens of interacting neighbors. Furthermore, there is no obvious way of reducing the number of lock/unlock operations.

For this purpose, blocking is done at the grid level by re-ordering the pair-list. The new order tries to ensure that whenever a certain interaction is processed, also other interactions of these atoms are processed shortly afterwards (see Figure 22.3). It is possible to determine a re-ordering that shows a working-set of fixed size, e.g. that of a (software) cache. To do so, atoms that have been touched once during the grid-traversation only favor the inclusion of other interactions of the same atoms for a short period of time afterwards. Its length is determined according to the desired working-set size. Keeping the data of the working-set atoms in temporal variables results in a reduction of atom data loads of about 20 for a working cache-size of 256. This temporal variables represent a *software cache*.

To decrease the number of lock/unlock operations, elements are purged from and loaded into the software cache not individually, but in groups. This already helps a lot, but it turned out that it is not yet sufficient. Ensuring mutual exclusion with a single lock results in considerable contention for that lock. The re-ordering of the pair list to achieve locality can also be exploited to reduce lock contentions. The idea is to assign different locks to nearby

no. charge-groups	13,824
pair-list length	442,484
orig. no. of purge/load operations of charge-group data operations	13,824 + 442,484 = 456,308
software cache size	256
no. of locks	16
optimized no. of purge/load operations of charge-group data	49,154
unoptimized no. lock/unlock operations	49,154
optimized no. lock/unlock operations	2,870

Table 22.1. Characteristic quantities for a 13,824 water molecule ensemble.

nodes in the grid (those that are geometrically adjacent or connected by short paths in the grid). Such atoms are simply those that are successively traversed in the re-ordered pair-list. Using e.g. 16 locks, which is a quantity that allows a good scaling behavior in terms of lock contention for reasonable degrees of parallelism, it was possible to reduce the number of lock/unlock operations to just one for about every 10 nodes that are jointly purged from the cache; see Table 22.1 for a summary.

Analogously to the pair-list construction module, the short-range force array is switched to a sharing mode when this module is entered and switched back to a replicating mode at its end to allow to keep other modules unchanged.

Load balancing. The parallelization as described so far results in load imbalance within the short-range interaction module. The reason is that the outcome of the pair-list construction module per process directly defines the work load of the short-range interaction evaluation module. The work load of the pair-list construction module has been scheduled for load balance, but the computational load of the pair-list construction process is not necessarily proportional to the amount of generated load for the succeeding short-range interaction evaluation module.

To eliminate load imbalance in the short-range interaction module, a redistribution of the process-local pair-lists is performed. Besides enforcing load balance, this allows even more optimizations. So far, the pair-list grid has been partitioned implicitly by the loop-scheduling within the pair-list construction module. Clearly, this results in a distribution in which each process' pair-list contains edges from the entire grid. The pair-list re-ordering step is able to deliver the more temporal locality the more geometrically adjacent grid regions are concentrated within single processes. Such a distribution is enforced at the same time the pair-lists are re-distributed for load-balancing reasons. This is done with a simple geometrical partitioning of the three-dimensional solution domain (see e.g. [8] for grid partitioning).

	thrombin (in water)	water (large data set)
no. solute molecules	1	0
no. solute atoms	3,078	0
no. solute charge-groups	1,285	0
no. solvent molecules (H_2O; = no. solvent charge-groups	5,427	13,824
no. solvent atoms	16,281	41,472
total no. of atoms	19,359	41,472
pair-list update / long-range force evaluation rate (t_{pl})	5	5
no. short-range solute-solute and solute-solvent charge-group interactions per time-step	57,735	0
no. short-range solvent-solvent charge-group interactions per time-step	175,326	442,584

Table 22.2. Characteristic quantities of the benchmark data sets.

22.4 Performance Results

22.4.1 Hardware Platform

The cluster used for evaluation purposes comprises six dual-processor Intel Pentium Pro machines (200 MHz; 256 kByte L2-cache) running under Windows NT 4.0. These, together with a file-server, are interconnected with Fast Ethernet for access to a common file system. For parallelization purposes, the machines are memory-coupled with Dolphin's first generation PCI-SCI adapters (Chapter 3).

22.4.2 Results

The performance of the parallelized code is evaluated using two benchmark problems from Biomos, the distributor of GROMOS [7]: a thrombin protein molecule in water and a large water ensemble (for parameters of the data sets, see Table 22.2). The thrombin data set is of interest, because it is the only one of relevant size that captures also long-range interactions. The water data set was chosen because its size is comparable to problems of real interest.

Figure 22.4 shows the speed-up when running the program on up to four machines, corresponding to a total of eight processors. The data has been derived by comparing the measured run-times to that of the original sequential code—not to the parallelized code running with a single process. However, both perform similar.

The non-linear speed-up has three major reasons:

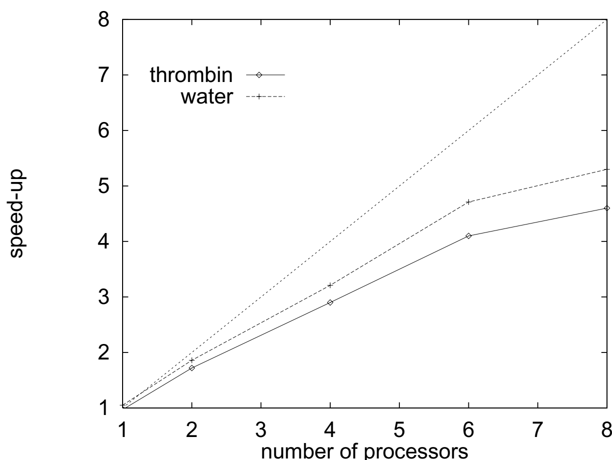


Fig. 22.4. Speed-up figures of the parallelized code on the SCI cluster.

1. The fraction of global shared memory that is local to each process decreases with an increasing number of machines. So, the fraction of remote memory accesses grows proportional to the number of machines.
2. The relative overhead of switching between sharing and replication of the shared memory regions grows, as the data has to be replicated on more compute nodes.
3. According to Amdahl's law, the remaining sequential parts of the program limit scalability for higher degrees of parallelism.

All three issues result from the strategy to limit the code modifications to just a few functions and to parallelize just the most time-consuming modules. This approach has been chosen with consciousness and it has payed off.

22.4.3 Performance Comparison to Other Parallel GROMOS Implementations

It is interesting to compare our performance figures to other parallel GROMOS implementations. Biomos itself presents some results of a thread-based shared memory version of GROMOS96 [7] on a SGI Power Challenge. Figure 22.5 compares the speed-up on the Power Challenge to the speed-up on the SCI-cluster.

Within the Europort project [6, 11], different but comparable data sets have been used, ranging from ∼2,000 to ∼30,000 atoms (proteins with water). The evaluation of the resulting message passing code has been done on various parallel machines, e.g. IBM SP1/2, Intel i860 Hypercube and SGI Power Challenge. For comparison purposes, the speed-up for 8 processors is of interest. Values between 4.4 and 5.6 are reported in [6] and between 4.8

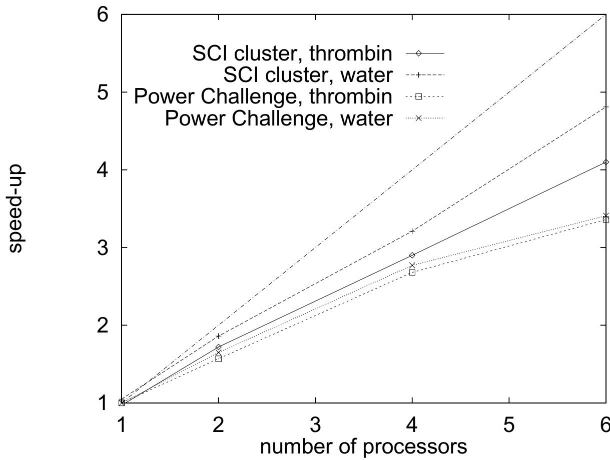


Fig. 22.5. Comparison of speed-up figures for the thrombin and the water benchmark data set for the Biomos parallelization on a SGI Power Challenge and the parallelization on the SCI-cluster.

and 6.7 in [11]. The first one refers to a not entirely parallelized code, comparable to the here presented implementation, while the later refers to a fully parallelized code.

The National Center for Supercomputing Application provides performance figures for UHGROMOS [12] on various platforms (HP/Convex SPP1200 and SPP2000, SGI Challenge and Origin). The reported speed-ups for comparable data sets of 10,000 to 15,000 atoms (protein in water) employing 8 processors range from 4.3 to 5.4. However, the speed-up is calculated on the basis of the computation time of the parallel version, running on a single processor. But this code is already about 15% slower than the original one.

All benchmarks were run on dedicated—and therefore expensive—parallel machines. Some of them are pure message passing machines like IBM SP1/2 and Intel i860 Hypercube, others are UMA (uniform memory access) machines with a more sophisticated memory system like SGI Challenge and Power Challenge, and still others belong into the class of CC-NUMA (cache-coherent non-uniform memory access) like HP/Convex SPP1200, SPP200 and SGI Origin. Nevertheless, the scaling behavior is not better than that of the parallel version on the SCI cluster.

22.5 Conclusion

The described work is part of a larger effort that aims at application parallelization on NUMA shared memory cluster systems. Here, methodologies are sketched that allow to exploit the given architecture for shared memory

parallelization. At a first glance, one might get the impression that the parallelization effort is quite high. But most of the work was spent to improve temporal data locality. Since the performance of the processors grows more rapidly than that of memory access, this is also of advantage for a pure sequential program. Other studies also report the necessity to improve per-process data locality as a precondition for a scalable parallelization [2].

All of the advantages of shared memory parallelization could also be achieved on the present cluster platform:

– the possibility of a *scalable* step-by-step parallelization process and
– the common view of all data for all processes which makes parallel programming much simpler than dealing with partitioned and distributed data structures as usually within a message passing programming model.

For the described parallelization of GROMOS96, just seven source code modules had to be touched. Five of them saw only minor modifications:

– three for parallelism and shared memory initialization purposes (∼6,500 lines of code) and
– two for I/O adoption (∼2,700 lines of source code).

Just two have been modified more extensively, these are where the actual parallelization took place:

– the pair-list construction with long-range force evaluation (∼2,200 lines of code) and
– the short-range force evaluation (∼2,300 lines of code).

The resulting performance of the parallel code is comparable to that of other parallelization efforts that employed expensive dedicated parallel machines. Considering the hardware costs and the time spent for the parallelization, shared memory programming on SCI-clusters is very attractive.

The experience gained during this work leads to the conclusion that NUMA shared memory cluster platforms are in fact more than just interesting alternatives to dedicated parallel systems as well as to LAN-connected cluster systems (e.g. PC clusters of the Beowulf type [17]). Lessons learned during this work are that there is a considerable demand for suitable programming interfaces and that it is essential for a programmer to be familiar with the NUMA performance characteristics.

References

1. M.P. Allen and D.J. Tildesley: *Computer Simulation of Liquids.* Oxford University Press, 1987.
2. H. O. Bugge and P. O. Husoy: Efficient SAR processing on the Scali System. *Proc. IPPS*, 1997.

3. T.W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott: Parallelizing Molecular Dynamics using Spatial Decomposition. *Proc. Scalable High Perf. Comp. Conf.*, 1994.
4. M. Dormanns, S. Lankes, T. Bemmerl, G. Bolz and E. Pfeiffle: Parallelization of an Airline Flight-Scheduling Module on a SCI-Coupled NUMA Shared-Memory Cluster. *Proc. High Perf. Computing Systems and Applications (HPCS)*, Kingston, Canada, 1999.
5. M. Dormanns, W. Sprangers, H. Ertl, and T. Bemmerl: A Programming Interface for NUMA Shared-Memory Clusters. *Proc. High Performance Computing and Networking (HPCN)*, pp. 698-707, LNCS 1225, Springer, 1997.
6. D. G. Green, K. E. Meacham, and F. van Hoesel: Parallelization of the molecular dynamics code GROMOS87 for distributed memory parallel architectures. *Proc. High Performance Computing and Networking (HPCN)*, pp. 875-879, LNCS 919, Springer, 1995.
7. GROMOS96 benchmark results: http://igc.ethz.ch/gromos/benchmark.html
8. H.-U. Heiß and M. Dormanns: Partitioning and Mapping of Parallel Programs by Self-Organization. *Concurrency: Practice & Experience*, Vol. 8, No. 9, pp. 685-706, Nov. 1996.
9. M. S. Lam, E. E. Rothberg, and M. E. Wolf: The Cache Performance of Blocked Algorithms. *Proc. 4th. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, 1991.
10. J. A. Lupo: Benchmarking UHGROMOS. *Proc. 28th Int. Conf. on System Sciences*, pp. 132-141, 1995.
11. K. Meacham and D. Green: Parallelization of the GROMOS87 Molecular Dynamics Code: An Update. *Proc. High Performance Computing and Networking (HPCN)*, pp. 170-176, LNCS 1067, Springer, 1996.
12. National Center for Supercomputing Applications: Computational Biology Applications, UHGROMOS and Gromos87 Benchmarks. http://mithril.ncsa.uiuc.edu/SCD/straka/PerfAnalysis/Apps/cb.html
13. S. M. Paas, M. Dormanns, T. Bemmerl, K. Scholtyssik, and S. Lankes: Computing on a Cluster of PCs: Project Overview and Early Experiences. *1. Workshop Cluster Computing*, Technical Report CSR-97-05, TU Chemnitz, Dept. of Computer Science, 1997.
14. M. Stockhausen: Parallelisierung und Evaluation eines Rechenkerns einer Erdbebensimulation auf einem speichergekoppeltem PC-Cluster. Diploma Thesis (in German), Chair for Operating Systems, RWTH Aachen, 1999.
15. S. Tholen: Parallelisierung raumakustischer Simulationsalgorithmen für SCI Cluster. Diploma Thesis (in German), Chair for Operating Systems, RWTH Aachen, 1998.
16. W. F. van Gunsteren, S. R. Billeter, A. A. Eising, P. H. Hünenberger, P. Krüger, A. E. Mark, W. R. P. Scott, and I. G. Tironi: Biomolecular Simulation: The GROMOS96 Manual and User Guide. BIOMOS b.v., Zürich, Groningen and VDF Hochschulverlag AG an der ETH Zürich, 1996.
17. M. S. Warren, D. J. Becker, M. P. Goda, J. K. Salmon, and T. Sterling: Parallel Supercomputing with Commodity Components. *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pp. 1372-1381, 1997.

23. SCI Prototyping for the Second Level Trigger System of the ATLAS Experiment

A. Belias[1], A. Bogaerts[2], D. Botterill[1], J. Dawson[3], E. Denes[2], F. Giacomini[2], R. Hauser[2], C. Hortnagl[2], R. Hughes-Jones[4], S. Kolya[4], D. Mercer[4], R. Middleton[1], J. Schlereth[3], P. Werner[2], F. Wickens[1]

[1] Rutherford Appleton Laboratory, Didcot (UK)
email: {a.belias,d.botterill,r.middleton,f.wickens}@rl.ac.uk
http://hepwww.rl.ac.uk/

[2] CERN, Geneva (Switzerland)
email: {Andreas.Johannes.Bogaerts, Francesco.Giacomini, Ervin.Denes, Reiner.Hauser, Christian.Hortnagl, Per.Werner}@cern.ch
http://www.cern.ch/

[3] Argonne National Laboratory, Illinois (US)
email: {jwd,jls}@hep.anl.gov
http://www.hep.anl.gov/

[4] University of Manchester, Manchester (UK)
email: {r.hughes-jones,d.mercer}@man.ac.uk, scott@a3.ph.man.ac.uk
http://www.hep.man.ac.uk/

23.1 Introduction

SCI's ultra low latency and high bandwidth [11] make it a possible candidate to solve the problem of interconnecting a large number of processors in order to filter data produced in high energy physics experiments. In particular this chapter focuses on prototyping for the second level trigger system of the ATLAS experiment [2], at the European Laboratory for Particle Physics (CERN).

After a brief introduction to the problem of real-time data selection in HEP experiments in Section 23.2, Section 23.3 presents the definition, design, implementation and test of a low-level Application Programming Interface (API) for SCI. In Section 23.4 and 23.5 the higher-level application software, the overall architecture of the system and their evolution are described.

23.2 The ATLAS Trigger System

Much evidence indicates that new physics phenomena may be visible in processes of energy scale around a few TeV (1 TeV = 10^{12} electron volts). The *Large Hadron Collider (LHC)* (now under construction at CERN, near Geneva, Switzerland) will bring protons into head-on collisions at these energies. The LHC will occupy the 27 km tunnel currently used for the *Large Electron Positron collider (LEP)*, and will allow scientists to penetrate deeper into the structure of matter than has previously been possible.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 397-414, 1999
© Springer-Verlag Berlin Heidelberg 1999

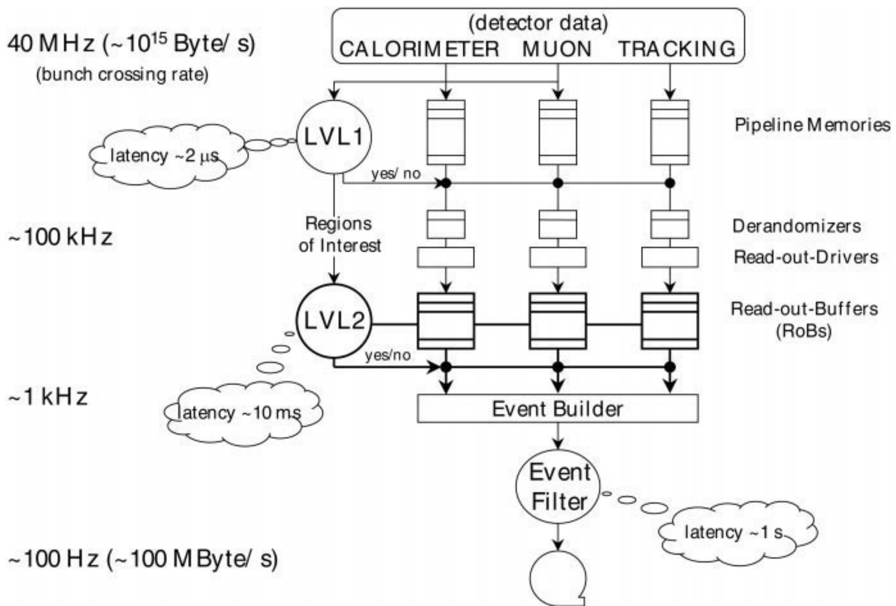


Fig. 23.1. Overall view of the ATLAS trigger system.

ATLAS is a large particle detector under construction to be ready for operation at the LHC in the year 2005. Its purpose is to measure the characteristics of the particles, such as their energies and trajectories, resulting from the proton-proton collisions (events). It consists of some 10^7 electronic channels, each of which must be read out for processing and recording.

The counter-rotating bunches of protons inside the LHC will pass through each other inside ATLAS at the rate of 40 MHz (i.e. every 25 ns). From simulation and knowledge of detector performance this is estimated to result in a data rate of 10^{15} byte/s. Clearly, it is way beyond any current technology to record this much data for subsequent analysis.

Thus, a radical approach is proposed whereby a mixture of custom hardware and general-purpose processors will be deployed in a highly parallel manner to reduce the amount of data coming out from the detector. The reduction comes from real-time rejection of un-interesting events, so that an event rate of about 100 Hz (corresponding to a data rate of about 100 MByte/s) is finally committed to long term store for subsequent off-line analysis.

The real-time selection process is termed "triggering". In ATLAS it is structured in three logical levels, as shown in Figure 23.1.

The first level (LVL1) must accept the 40 MHz beam-crossing rate. This rate can be handled only by specialized electronics operating on coarse grain data from a few sub-detectors; the level-1 processors implement simple local, highly parallel algorithms. The decision latency is limited to about 2 µs by the size of the on-detector pipeline buffers. The maximum LVL1 output rate

is designed to be about 100 kHz and thresholds would be adjusted to meet this requirement.

The second level of the trigger system (LVL2) must reduce the trigger rate to a level of about 1kHz which can be sustained by the event building system, which collates fragments from all sub-detectors into complete events. Full granularity, full precision data for an event would be available to the LVL2. However, in order to save processing power and network capacity, it only examines regions of the detector identified by LVL1 as containing interesting information (Regions of Interest or RoIs).

The third and last level of the trigger system is the Event Filter (EF), which takes the definitive decision if an event has to be kept or discarded. It operates on complete events, possibly using off-line analysis code.

SCI-based work has concentrated on the second level trigger which, in its final version, is estimated to consist of about 2000 data sources and 1000 processing nodes, all connected by a high performance network that should sustain a traffic rate of several GByte/s.

23.3 Low-Level API

To allow different hardware implementations and software emulation of incomplete hardware functionality, IEEE Std 1596.9 "Physical layer Application Programming Interface for the Scalable Coherent Interface (SCI PHY-API)" [12] defines an API to abstract the underlying SCI physical layer (see Chapter 10).

Initial SCI activity, prior to commercial support for SCI software, concentrated on an in-house implementation of the SCI PHY-API. This partial implementation focussed mainly on giving the possibility to map remote memory in particular in the Demonstrator Programme (see Section 23.4).

More recently, CERN and RAL have been involved in a EU-funded project, named "Standard Software Infrastructures for SCI-based Parallel Systems" (SISCI) [16], aiming at developing highly advanced, state-of-the-art, software environments and tools to exploit the unique hardware capabilities of the SCI communications standard. One of the major objectives of the project was again the specification of a low-level API. However, in this case it represents an abstraction of both hardware and low-level software; in this respect it can be seen as an additional layer, more application-oriented, on top of the SCI PHY-API.

The SISCI API [9] supports:

Distributed shared memory whereby a memory segment can be allocated on one node and mapped in the virtual address space of a process running on another node. Data is then moved using programmed I/O.

DMA transfers to move data from one node to another without CPU intervention.

Remote interrupts whereby a process can trigger interrupts on a remote node.

Fault tolerance allowing a process to check if a transfer was successful and to catch asynchronous events, such as a link failure, and take appropriate action.

In the design phase of the SISCI API much emphasis was placed on protection against improper use of the low-level resources, but at the same time guaranteeing that data transfers could still be performed efficiently. The API design followed an object-oriented approach: the characteristics of the available resources (e.g. memory segments or interrupts) are encapsulated in data structures (called descriptors) that a client application can refer to only via a handle and access only via the API functions. Resources (and their corresponding descriptors) are arranged in a dependency graph, in such a way that it is not possible to free a resource if other resources, located either locally or remotely, depend on it. For many resources a finite state diagram has been defined. Changes of state may be caused by calls to API functions or by asynchronous events (e.g. a link failure).

The SISCI API also provides some functions, reserved for "expert" users, that allow access to low-level resources, such as CSR [13] registers.

23.3.1 Basic Performance Measurements

To understand the potential of this technology and its behavior in a complex system it is important to have a good knowledge of the performance of SCI under well defined conditions. For this reason a number of benchmarks have been defined and applied to a variety of configurations. All the measurements shown in this section are based on the remote shared memory paradigm.

Some benchmarks aim at understanding the characteristics of the particular SCI hardware, in our case the PCI-SCI adapter card from Dolphin Interconnect Solutions [7], based on their link controller LC-1 running at a link speed of 200 MByte/s [8]. For example Figure 23.2, left, shows how the bandwidth varies according to the size of the transfer (8 to 4096 bytes) and to the maximum number of allowed outstanding SCI requests (1, 2, 4 or 8):

– for a given number of allowed outstanding requests, the worst performance is obtained for a message size of 8 bytes; the throughput doubles for 16-byte transfers (since the number of 16-byte SCI packets is constant but the amount of transferred data doubles) and remains constant for 32-byte transfers (since both the number of SCI packets and the amount of transferred data double). For 64-byte data transfers, 64-byte packets are used, fully exploiting the SCI hardware and causing a sharp increase in the achieved bandwidth. For data transfers larger than 64 bytes the throughput remains constant because a double amount of data is transferred using a double number of SCI packets.

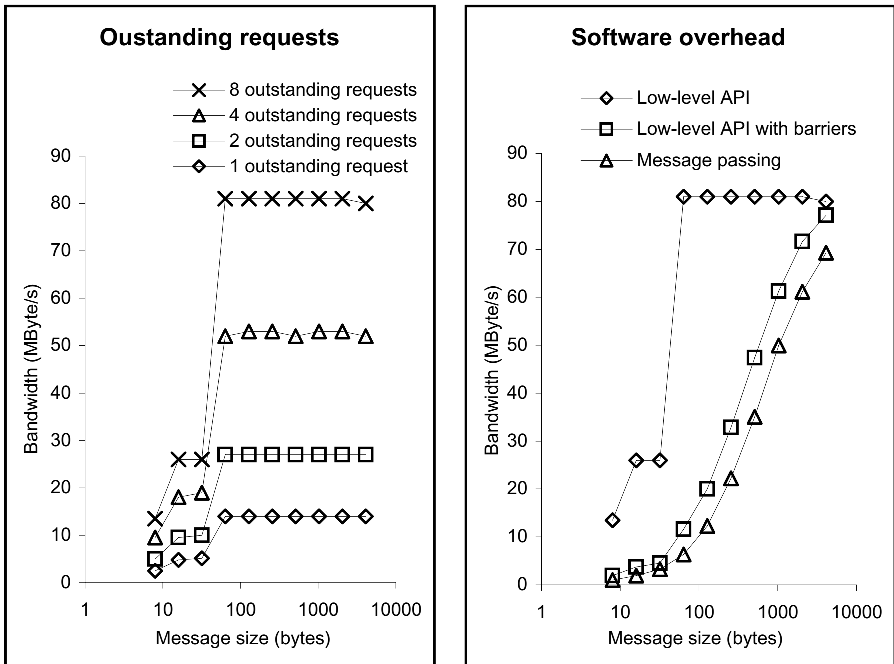


Fig. 23.2. Bandwidth as a function of message size, depending on the maximum number of allowed outstanding SCI requests (left) and with different software overheads (right).

– for a given size of the data transfer, the bandwidth doubles going from 1 to 2 and from 2 to 4 outstanding requests, thus showing perfect scaling. The scaling going from 4 to 8 outstanding requests is instead less than linear and is due to the fact that the limit for programmed I/O is reached on the host PCI bus [15].

Other benchmarks aim at estimating the degradation in performance when more and more flow control is included in the data exchange protocol. The protocol can simply require a barrier operation after each send to guarantee that data is not locally cached, or it can be more sophisticated and require, for example, acknowledgments from the receiver. A message-passing protocol has been defined, implemented on top of the remote shared memory paradigm provided by the low-level API and provides synchronization between source and destination. This follows a simple mechanism in order to communicate:

1. sender and receiver agree initial addresses for control word and start of data area;
2. the sender writes the data at a specified remote address on the receiver;

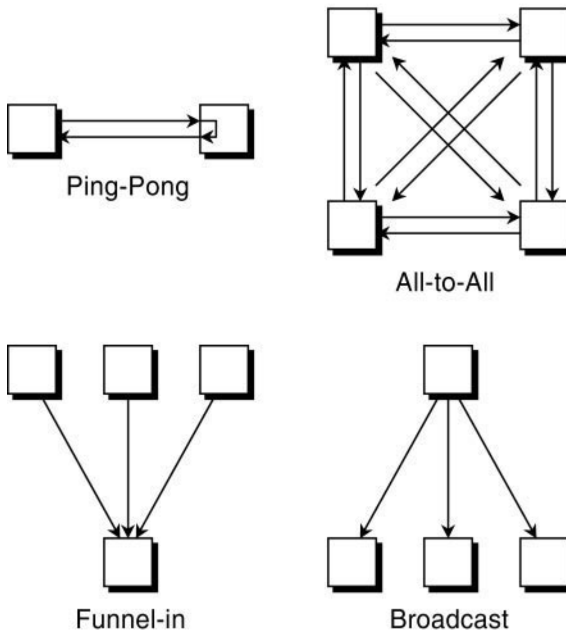


Fig. 23.3. Suite of technology independent communication benchmarks.

3. the sender writes a control word, representing the size of the message, at an agreed location on the receiver, to signal the message arrival;
4. the receiver sends back to the sender an acknowledgment, representing the next address to use for sending data.

Figure 23.2 (right) shows how the throughput decreases passing from the raw bandwidth (where data is written as fast as possible into the remote node) to the case where a barrier is used after each send, to the case where the message passing is used. The difference between them is especially evident for short data transfers (where barrier and synchronization overheads respectively dominate), whereas it becomes less important for larger messages.

Since other technologies, such as ATM and Ethernet, are possible candidates for the ATLAS second level trigger, a number of technology independent benchmarks, shown in Figure 23.3, have also been defined [1]. The purpose is to provide a common way to evaluate important networking aspects typical of the final system. These benchmarks include:

Ping-Pong where one node sends data to another node and waits for the data to be returned. Half the round-trip time is used to calculate the throughput.

Broadcast where one node sends data to several receivers. SCI does not support hardware broadcast which is thus simulated in software.

Funnel-in where several data sources send data to a single receiver.

All-to-All where all nodes send data to each other simultaneously. In an ideal system, increasing the number of nodes in the system increases the overall system throughput.

These benchmarks have been executed for varying message sizes and numbers of nodes and used the message-passing library mentioned above.

Figures 23.4 show the overall system throughput as a function of message size for Ping-Pong, All-to-All, Funnel-in and Broadcast.

For Ping-Pong data is shown for one and two pairs. The overall system throughput is clearly doubled by the addition of a second pair to the ring.

The Broadcast plot shows little variation as more data receivers are added to the system. The small visible variations are due to minor differences in processor motherboard versions.

In the case of Funnel-in, overall system throughput increases as nodes are added to the system, though it is clear that the addition of the fourth node causes significant reductions in throughput for larger message sizes. This is caused by the receiving node's inability to keep up with three senders. In particular, the host PCI bus appears to be a bottleneck and incoming SCI traffic is 'busied', with a consequential immediate retry on SCI [5].

In the All-to-All test, the increase from 2 to 3 nodes appears to give almost 3 times the performance boost, as expected, at least for smaller message sizes. However, above 1 kByte in the 3 node system, throughput degrades as the receiving node is no longer able to cope. In the 4 node system each node has to process data from three sources, in addition to sending to 3 destinations and is likely to be CPU limited. Thus there is little performance enhancement.

23.4 The ATLAS Level-2 Trigger Demonstrator

A Demonstrator Programme was defined to evaluate different network technologies that could satisfy the requirements of the second level trigger system.

The main components of the system are the following:

Read-out-Buffer (RoB) a RoB keeps fragments of events coming from a specific part of the ATLAS detector and represents a data source for the LVL2. Events are stored in the RoBs while they are being analyzed by the LVL2.

Feature Extractor (FeX) a FeX is responsible for the calculation of certain parameters (features) of a Region of Interest, extracting it from the data it has received from a few RoBs. Feature extraction is also known as "local processing".

Global processors a Global processor collects all of the features concerning one event and, based on these, decides if that event is of interest. The decision is then forwarded to the Supervisor for the appropriate action.

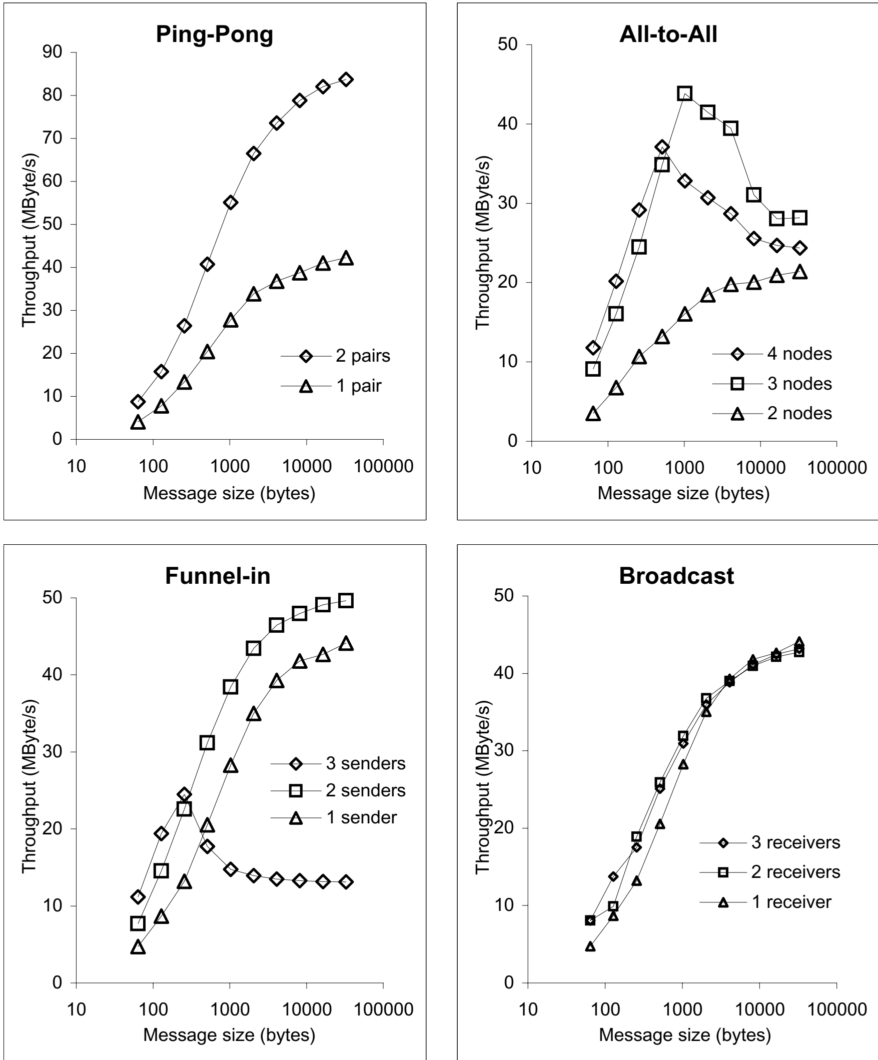


Fig. 23.4. Overall system throughput as a function of message size for Ping-Pong, All-to-All, Funnel-in and Broadcast.

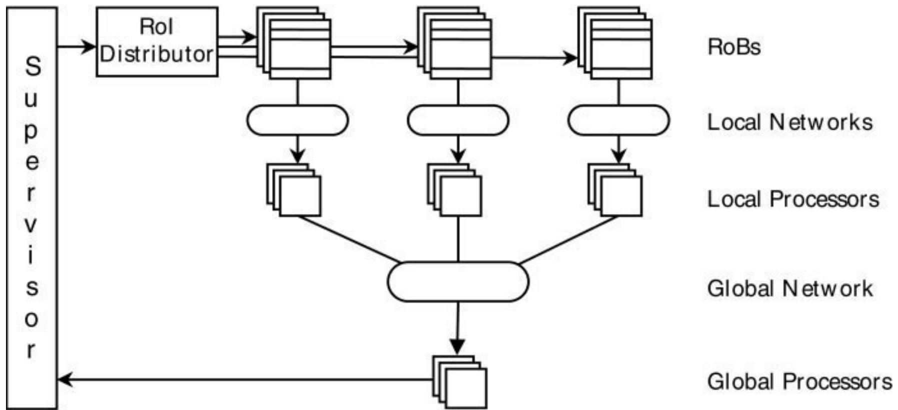


Fig. 23.5. The local-global option of the second level trigger system

Supervisor the Supervisor accepts event identifiers and RoI pointers from LVL1, dispatches events to LVL2 processors for appropriate examination and waits for decisions to come back. If the event is not interesting it is erased from the RoBs, otherwise the Supervisor informs the Event Filter to refine the analysis.

The demonstrator focusing on the SCI technology assumed a parallel push architecture for the system, as shown in Figure 23.5. Under the control of the Supervisor process [4], RoI pointers are distributed to selected RoBs. Each affected RoB pushes data to Feature Extractor (FeX) processors, which, in parallel for each RoI and for each detector layer in the RoI, determine specific characteristics of an event. The features from a single event are then passed to a Global processor, that combines them, generates a trigger decision and forwards it to the Supervisor, which then decides whether the event should be kept or discarded. The Supervisor communicates with the RoBs using an RoI Distributor. The parallel push architecture is also referred to as the "Local-Global" option [10].

To test both SCI technology and fundamental system design concepts, extensive studies have been carried out on a vertical slice of the second level trigger, i.e. a system constituted by all the elements that should appear in the final system but much reduced in number. The SCI vertical slice, shown in Figure 23.6, was set up at CERN in November 1997 and included 3 RoBs, 3 FeXes, 3 Globals and a Supervisor, connected by an SCI network and the RoI Distributor.

23.4.1 Hardware

The FeX and the Global processors were Alpha computers from DEC of different clock speeds (AXPpci33 at 166 or 233 MHz and Multia at 166 MHz)

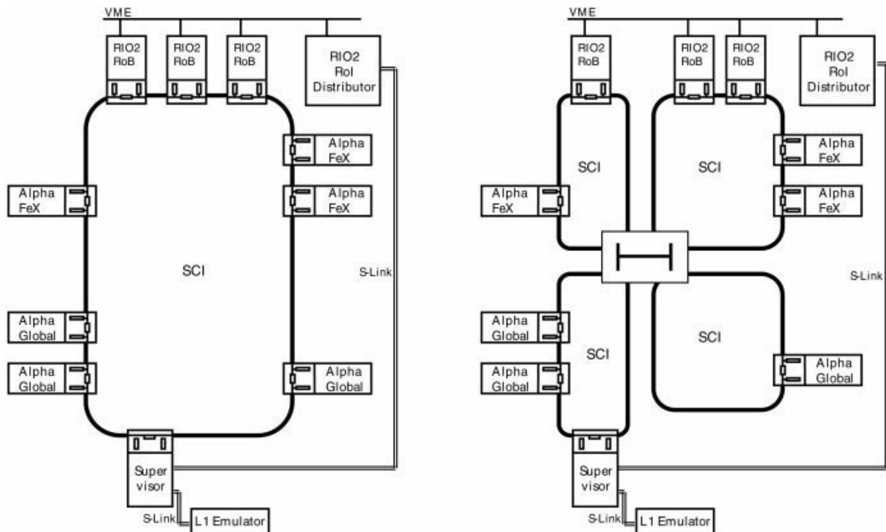


Fig. 23.6. Vertical slice configuration with the SCI network constituted of a single ring (left) or of four ringlets connected through a 4-port switch (right).

running μC/OS, a small stand-alone real-time kernel [14]. The RoB, the Supervisor and the RoI Distributor were VME-based RIO2s from CES (type 8061 and 8062 with a clock speed of 100 MHz and 200 MHz respectively) [6] running LynxOS, a real-time Unix operating system. All the processors, except the RoI Distributor, were equipped with Dolphin PCI-SCI adapters. The SCI network was configured both as a single ring and as four ringlets connected by a 4-port switch. The RoI Distributor was connected to the Supervisor via an S-Link [3] and to the RoBs via the VME backplane [17].

23.4.2 Software

The SCI hardware provided by Dolphin ICS offers several facilities to send and receive data. In the tests two modes were evaluated:

Remote Shared Memory where the sender maps into its address space a memory segment residing on a remote node and then transfers data transparently by a memory copy operation. On top of this, the message-passing protocol introduced in Section 23.3.1 has been used.

DMA and Ring Buffer where the sending CPU loads the DMA engine of the SCI adapter with a transfer specification. The adapter then fetches the data from memory and transmits it to the remote node, where it is placed in a ring buffer. Application software then extracts data from the buffer and processes it.

23.4.3 Vertical Slice Configurations

Two parameters were used to characterize the performance of the system: the event latency and the average time per event. The former is defined as the time from when the Supervisor injects an event into the system to the time it receives a decision back. The latter is the average time between two trigger decisions and is measured by dividing the duration of the whole test by the number of events that have passed through the system. The values of the two parameters differ considerably when more than one event is allowed in the system at the same time.

Several types of parallelism were present in the system (see Figure 23.7) and their impact on the time-per-event parameter was studied:

Pipelining whereby multiple events are allowed to enter the system quasi simultaneously, exploiting the inherent pipeline structure of the RoB - FeX - Global chain.

Event Parallelism with multiple RoB - FeX - Global chains running concurrently.

Fragment Building allowing several event fragments from different RoBs to be sent in parallel to the same FeX.

RoI Parallelism allowing several FeXes (each possibly receiving data from multiple RoBs) to analyze multiple RoIs of several detectors of the same event in parallel.

For each of the possible configurations, all in a closed loop under control of the Supervisor, latency and average time per event were measured varying the size of the data transfer between RoBs and FeXes and the number of events allowed in the system at the same time. In some cases an SCI switch was introduced to partition the network in smaller rings. The goal was to evaluate the impact of the different forms of parallelism on the performance of the system and to find where saturation would eventually occur.

Pipeline. A single stream is constituted by a RoB, FeX and Global; for this test each event has only one RoI and this RoI is contained in a single RoB. The stream is activated by the Supervisor when an RoI record of an event is received from LVL1 and is terminated when the Supervisor receives the trigger decision from the Global processor.

Since the stream has an intrinsic sequential structure with each stage corresponding to a processing step, several events can be pipelined in the system. Figure 23.8 (left) shows the time per event as a function of RoB data size for different numbers of events allowed in the stream.

For one event in the system the time per event is determined by the total loop latency. For two events, they are distributed over the stages with no queues forming (i.e. no increase in latency) until the data size is slightly over 2 kByte and the time per event just scales.

For longer events, or more than two allowed in the system, a queue forms at the slowest element and the rate is limited to the speed of this element. For

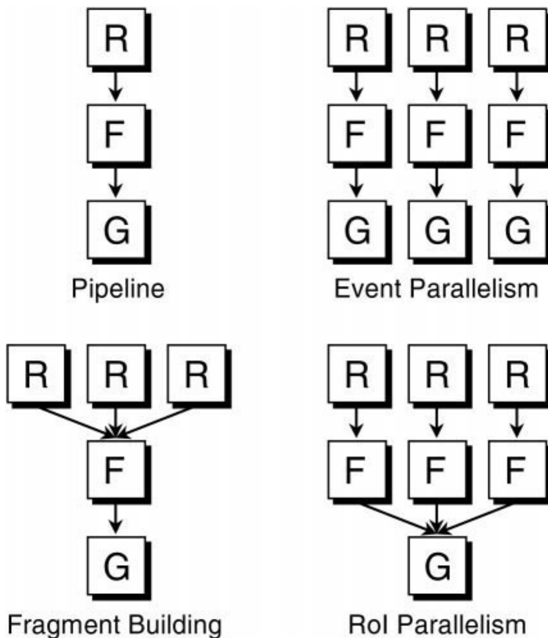


Fig. 23.7. Forms of parallelism studied in the SCI Demonstrator Programme.

most event lengths, the slowest element is the RoB-to-FeX transfer (with an effective bandwidth of 18-20 MByte/s), but for very short events it is the RoI Distributor. In addition, for these very short messages, there is a small but significant contention of the PCI bus of the RoB, arising from attempts to receive RoIs from the RoI Distributor (over the VME-PCI bridge) at the same time as initiating transfers to SCI by the RoB host CPU. This contention slows down the RoI Distributor as the number of SCI packets increases.

Event Parallelism. Scalability is one of the most important characteristics of the level-2 trigger system. With the available equipment it was possible to arrange up to three RoB - FeX - Global streams, controlled by a unique Supervisor. As in the previous case, each event has only one RoI and this RoI is contained in a single RoB.

In going from one to two and three streams one would expect a proportional increase in the aggregate bandwidth and a decrease in the time spent for each event. Although scaling has been observed, it is not perfect, owing to the fact that the Supervisor and the RoI Distributor are shared resources. The overall system performance is thus limited, especially for small message sizes where the operating frequency is higher.

RoI Fragment Building. The system was set up with one, two or three RoBs, one FeX and one Global, to evaluate the case where there is only one RoI per event, but the RoI is distributed over multiple RoBs. Thus, a FeX

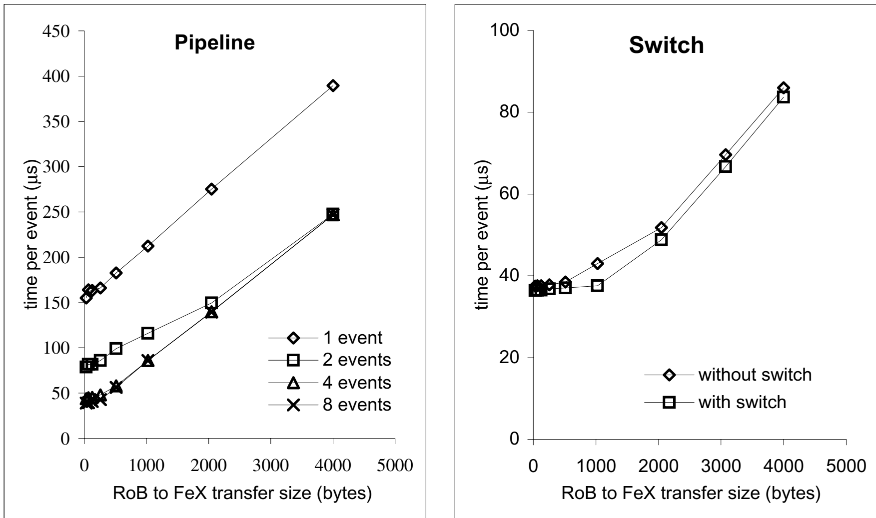


Fig. 23.8. Effect of event pipelining (left) and of the introduction of a switch (right) in the SCI demonstrator.

receives data from several RoBs, which has to be combined to build an RoI. This configuration tests the efficiency of the RoI fragment builder inside a FeX and the cost or benefit of spreading event data over several sources. Since the FeX has to wait for an RoI fragment coming from each RoB, the performance is affected by the degree of parallelism of the RoB-to-FeX transfers and the fragment building.

The degree of parallelism was observed to be poor, for two reasons:

– the RoB-to-FeX transfers are partly serialized: owing to the lack of a broadcast option in the VME bus, the RoI Distributor starts successive RoBs with a delay of about 15 µs between them.
– since the speeds of the processors are different, the FeX has to wait for the slowest RoB before completing an RoI.

RoI Parallelism. The system was composed of one, two or three RoB - FeX combinations feeding into one Global. An event contained one, two or three RoIs respectively, each in a single RoB. Since the Global has to wait for a feature coming from each FeX before taking a decision, the performance was similar to that of a single stream, but only if there was complete overlap of all the RoB - FeX combinations. Deviations were caused by the 15 µs delay of the RoI Distributor and the fact that the Global has to wait for the slowest of the RoB - FeX combinations. Nevertheless, considerable parallelism was seen.

Switch. The configuration used to evaluate event parallelism with three independent streams controlled by the same Supervisor has been used to study

the impact of a 4-port switch on the performance of the system. The nodes were arranged in four ringlets each connected to a switch port, as shown in Figure 23.6 (right).

As shown in Figure 23.8 (right) the switch leads to a small improvement in the performance of the system, despite the extra delay of about 1.5 μs in the packet latency. The improvement could be attributed to the following:

– there were less nodes on each ringlet connected to the switch and this reduced the time to send a packet around the ring from a RoB to a FeX.
– there was less traffic on each ringlet reducing the chance of any delay.

23.4.4 Conclusions

The tests have demonstrated high rate operation of the components used in this system. In all cases, except the RoI Distributor, the rates are comparable to those required, albeit with simplified functionality.

Some scalability in terms of pipelining, event parallelism, fragment building and RoI parallelism has been demonstrated for typical ATLAS data fragment sizes (around 1 kByte). Two bottlenecks have been identified in the system that limited scalability: the RIO2 PCI bus bandwidth and the RoI Distributor rate.

23.5 Objectives and Design of the Second Prototype

When the first prototyping was started as part of the ATLAS Demonstrator Programme, it was not at all clear if a second level trigger system could be built from commercial components. The main emphasis was therefore on exploring the hardware limits of networks and their interfaces, in particular bandwidth and latencies for transmission of relatively small messages, typically about 64 bytes for control and about 1 kByte for data. As confidence grew it became apparent that larger more uniform systems with more realistic data patterns allowing more detailed measurements were necessary for a better understanding.

23.5.1 Lessons Learned from the Demonstrator

The deeper analysis of the ATLAS level-2 trigger problem during the Demonstrator Programme showed that there were significant advantages, in lower bandwidth and processor requirements, if a sequential selection strategy were used. Furthermore the tests of different hardware increased confidence that commercial components would be able to provide the performance required for a system to be built using a single network interconnecting the different parts. Whilst the separate distribution path used for data requests in the SCI

demonstrator could still be used with a sequential selection strategy, it is an unnecessary complication. Thus the preferred solution for the testbeds in the next phase assumed sequential selection and a single network.

Testbeds comprising 32 or more processors running in real-time are already complex and need good quality software, with flexible configuration and control capabilities. The software from the Demonstrator was optimized to get good performance from the hardware, but was inadequate to use in the larger systems planned in this next phase. Understanding the behavior of a complex system needs detailed measurements of the behavior of hardware and software components. The analysis of an inhomogeneous system consisting of dissimilar components such as processors running at different clock speeds or different I/O interfaces already gave difficulty in the demonstrators and should be avoided in the testbeds if possible. In the Demonstrator selection algorithms were replaced by idle loops. But a more realistic environment with real algorithms using simulated detector data would give a more complete test of the system.

23.5.2 Testbed

The available time, nine months to design and build the system and one year for measurements and analysis dictates the need for a simple, common 'reference' hardware and software platform. The reference testbed hardware consists of PCs connected to a single (Fast)Ethernet network. RoBs and Supervisor are emulated with PCs.

The reference software is designed to be platform independent; actual implementations are foreseen for PCs running either Windows NT or Linux. These operating systems whilst widespread are sufficiently different to ensure a high degree of platform independence. TCP and UDP are used for basic communication.

The testbed is a generic system which is expected to have neither the performance nor the size required for the final ATLAS system. It serves as an environment to study trigger selection procedures, faster networking, scalability and architectures. More realistic systems can be derived from the generic testbed by substituting especially optimized hardware and software components. This allows for a better evaluation of new technologies when they become available in a standardized environment at minimal cost.

Architectures being considered at the moment are either clusters of single-processor or small multi-processor machines interconnected by a network or fully commercial HPCN systems. It is not yet clear if sufficiently large clusters with enough I/O bandwidth and network switching capacity operating at the design trigger frequency of 100 kHz will be available commercially. Demands on the network can be reduced by decomposing it into smaller subnets.

A generic model for the flow of data which is common to all architectures is illustrated in Figure 23.9. The operation is as follows: pointers to RoIs in the different detectors as identified by LVL1 are reformatted by the

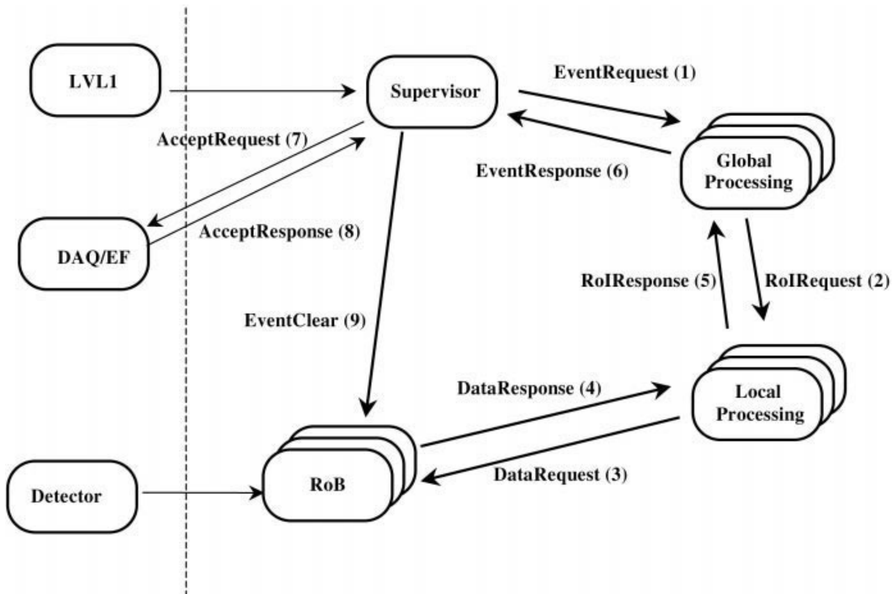


Fig. 23.9. Logical architecture of the second level trigger system. Left of the dotted line are the interfaces to the detector, level-1 trigger and data acquisition.

Supervisor and sent to Global Processing Units as an EventRequest. These need the results ("features") calculated from the detector data contained in a small window identified by the RoI pointers. This calculation is delegated to FeX Local Processing Units by sending RoIRequests. Feature Extraction Processes send DataRequests to the RoBs. Raw data flows back to the Feature Extractors (DataResponse), features to the Global Processors (RoIResponse) and finally a trigger decision to the Supervisor (EventResponse).

The generic data flow model allows a common software basis for a variety of architectures which differ in the way algorithms and data are distributed and accessed across the network. This can be a single PC, though larger architectures require a network. The single farm solution consists of a single switching network interconnecting Supervisor, RoBs and a farm of processors executing algorithms on a per event basis. The Local-Global solution used in the SCI Demonstrator (see Section 23.4) sub-divides the processing farm and now uses a "requested push"; it would allow the partitioning of the system, and hence of the network, into several RoB-FeX sub-farms, one per sub-detector, and one Global sub-farm. Commercial versions using HPCN systems, although not currently competitive, may in the future offer a useful solution based on the Local-Global option.

23.5.3 Software

Object Oriented design and implementation techniques have been adopted for the software which falls roughly in two categories:

Algorithms which are strongly physics oriented;
Framework software for the organization of the flow of data.

Algorithms comprise feature extraction and global processing (or steering), which transforms features into physics signatures (such as the identification of particles and their properties). The algorithms have been redesigned or adapted to an OO framework and will be run in testbeds using simulated detector data which is pre-loaded in (emulated) RoBs. One of the challenges of the project is to engineer algorithms such as to make them independent of the architecture, i.e. unaware of the fact that both data and algorithms may be distributed across processor boundaries.

To make the software independent of the operating system a small number of services (a subset of those offered by e.g. POSIX and Windows NT) have been encapsulated in OS Services Interface Objects. Similarly to ensure the possibility of testing an open-ended range of communications technologies (currently including ATM, Fast and Gigabit Ethernet, SCI and vendor specific HPCN technologies) only the higher levels of message passing based protocols have been defined, supplemented by a low-level implementation using TCP or UDP for the purpose of testing the complete software chain. The message passing layer is intended to leave enough flexibility to optimize an actual implementation for either message passing or shared memory based interconnects. Message passing protocols are further encapsulated in objects which may use RPC-like techniques to hide the potentially distributed nature of algorithms and data to the application. Additional support software being developed comprises tools and APIs for access to databases, process management, displays, monitoring and run control.

23.5.4 SCI Testbed

An SCI-based prototype of the ATLAS second level trigger system has been constructed. The software supports both simple message passing over the SISCI API and MPI. Our system consists of 16 dual Pentium II PCs equipped with Dolphin PCI-SCI adapters based on their 400 MByte/s technology and connected by a 16-port SCI switch. The resulting measurements will be used as input for a Technical Proposal of the ATLAS High-Level Trigger, Data Acquisition and Detector Control System due at the end of 1999.

Acknowledgments

This work has been partially funded by the SISCI Project (EU Contract 23174). We would also like to thank our many colleagues within ATLAS and especially those in the Trigger/DAQ group.

References

1. J. Apostolakis et al., Abstract Communication Benchmarks in Parallel Systems for Real-Time Applications, *CHEP '97*, Berlin, April 7-11, 1997.
2. ATLAS Collaboration, Technical Proposal for a General-Purpose Experiment at the Large Hadron Collider at CERN, CERN/LHCC/94-43, Geneva, Switzerland, December, 1994.
3. H.C. van der Bij et al., S-LINK, a Data Link Interface Specification for the LHC Era, presented at the *X IEEE Real-Time Conference*, Beaune, France, September 22-26, 1997.
4. R. Blair et al., The ATLAS Level-2 Trigger Supervisor, presented at the *2nd Workshop on LHC Electronics*, Balatonfüred, Hungary, September 1996.
5. A. Bogaerts et al., Studies of SCI for the ATLAS Level-2 Trigger System, *X IEEE Real-Time Conference*, Beaune, France, September 22-26, 1997.
6. Creative Electronic Systems, RIO II User Guide, Geneva, Switzerland.
7. Dolphin Interconnect Solutions, PCI-SCI Cluster Adapter Specification, 1996.
8. Dolphin Interconnect Solutions, Link Controller LC-1 Specification, 1995.
9. F. Giacomini et al., Low-level SCI Software: Requirements, Analysis and Functional Specification, Dolphin ICS, Oslo, Norway, May, 1998.
10. J. R. Hansen, Local-Global Demonstrator Programme for the ATLAS Second Level Trigger, presented at the *X IEEE Real-Time Conference*, Beaune, France, September 22-26, 1997.
11. IEEE Computer Society, IEEE Standard for Scalable Coherent Interface (SCI), IEEE Std 1596-1992, August, 1993.
12. IEEE Computer Society, Physical Layer Application Programming Interface for the Scalable Coherent Interface (SCI PHY-API), IEEE Std P1596.9, Draft 0.51, June 15, 1997.
13. Information technology - Microprocessor systems - Control and Status Registers (CSR) Architecture for microcomputer buses, ISO/IEC 13213, ANSI/IEEE Std 1212, first edition 1994-10-05.
14. J. J. Labrosse, µ*C/OS - the Real-Time Kernel*, R&D Publications Inc, Distributed by Prentice-Hall, ISBN 0-13-031352-1, 1992.
15. PCI Local Bus Specification, Rev 2.1, PCI Special Interest Group P.O. Box 1470, Portland, OR 97214 USA.
16. SISCI, Standard Software Infrastructures for SCI-based Parallel Systems, ESPRIT Project 23174, 1997.
17. VME64 ANSI/VITA 1-1994, published by VMEbus International Trade Association, Scottsdale, AZ 85253, USA.

Part VIII

Tools for SCI Clusters

When planning the purchase of a compute cluster, usually much thought is spent on the choice of compute nodes, interconnects, switches, and—to a lesser extent—the operating system and system software. Important software tools for system configuration, user administration, fault tolerance, debugging, and monitoring are often overlooked. While in small systems, this does not matter too much, the lack of suitable software tools might become a nightmare, though, when trying to operate compute clusters for a large, diverse user community. The following three chapters deal with tools.

In Chapter 24, researchers from Technische Universität München (TUM) present a network monitoring tool that has been implemented in the context of their SMiLE project. With the data obtained from a hardware monitor on their own adapter card (see Chapter 4), the TUM researchers have implemented an infrastructure for the evaluation and controlled deterministic execution of hardware-supported distributed shared memory architectures.

Based on the Dolphin PCI adapter cards, researchers from the University of Paderborn have developed a simple but powerful software that allows the user to observe the utilization of processors and the network. The software monitor presented in Chapter 25 is intended for administrators to trace the system status and for users to debug and tune their application. In contrast to the above TUM project, this monitor does not actively influence the application.

Finally, Chapter 26 addresses the important issue of operating large SCI clusters as general purpose compute servers in a multi-user environment. The authors from Paderborn present the architecture of their *Computer Center Software (CCS)* which provides mechanisms for system partitioning, job scheduling, and user access management. With CCS, an SCI cluster is no longer seen as a collection of machines, but rather as a dedicated high-performance computer. Hence the focus of CCS is on supporting parallel high-performance applications rather than throughput computing (which is the prevalent operation mode for LAN clusters).

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, p. 415, 1999
© Springer-Verlag Berlin Heidelberg 1999

24. SCI Monitoring Hardware and Software: Supporting Performance Evaluation and Debugging

Wolfgang Karl, Markus Leberecht, Michael Oberhuber

Lehrstuhl für Rechnertechnik und Rechnerorganisation – LRR
Institut für Informatik der Technischen Universität München
80290 München, Germany
email: {karlw,leberech,oberhube}@in.tum.de
http://wwwbode.informatik.tu-muenchen.de/

24.1 Introduction

The development of a parallel program which runs efficiently on a parallel machine is a difficult task and takes much more effort than the development of a sequential one. A programmer has to consider communication and synchronization requirements, the complexity of data accesses, as well as the problem of partitioning work and data, depending on the underlying programming model. Additionally, the potentially nondeterministic behavior of concurrent activities running on the parallel machine aggravates the test and debugging phase in the software development cycle. Even when a program is validated and produces correct results, a considerable amount of work has to be done in order to tune the parallel program to efficiently exploit the resources of the parallel machine.

This task becomes even more complicated on architectures supporting fine-grained execution such as PCs clustered with novel high-speed, low-latency networks like the Scalable Coherent Interface (SCI). SCI supports memory-oriented transactions over a ringlet-based network, effectively providing a global virtual bus. Remote read latencies are on the order of $5\,\mu s$ for I/O bus-based SCI adapters, as is demonstrated for the LRR-TUM adapter in Chapter 4 as well as in Chapter 3 for the Dolphin adapters. Through these properties, SCI implements a hardware-supported distributed shared memory (DSM) system on a network of PCs. On this class of architectures, communication events cannot be observed easily by appropriate tools since they are potentially very frequent, comparably short, and cannot be easily distinguished from local memory reads and writes.

The SMiLE system (Shared Memory in a LAN-Like Environment) [4] belongs to this class of architectures and represents a network of Pentium-II PCs clustered with the SCI interconnect. Its NUMA characteristics (non-uniform memory access) is implemented in hardware and is based on a custom PCI/SCI adapter described in Chapter 4 that plugs into the PC's PCI local bus. A hardware monitor as part of an event-driven hybrid monitoring system

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 417-432, 1999
© Springer-Verlag Berlin Heidelberg 1999

for the SMiLE PC cluster is able to deliver detailed information about the run-time and communication behavior of parallel programs. This information can be utilized by tools for performance evaluation and tuning as well as debugging [6] [5]. The hardware monitor is being implemented as a second PCI card and attached to the PCI/SCI adapter side-by-side.

The controlled deterministic execution approach *codex* [3] provides a generic method to overcome the problems arising from the nondeterministic behavior of parallel programs during the test and debugging phase. It is based on POEM (*Parallel Object Execution Model*) [3], a framework for modeling parallel execution independently from the underlying programming model. This allows the specification of the requirements for a deterministic execution of test cases. While for message passing *codex* can be implemented in a fairly straightforward way by instrumenting messaging layers, this is not possible for the DSM-oriented execution of the type of architecture mentioned above.

This chapter deals with our approach to deliver run-time information to tools for performance analysis, and to integrate controlled deterministic execution into the hardware-supported DSM execution paradigm provided by the SMiLE PC cluster. We will not focus on a particular programming model. However, remote memory transactions are considered to be the base of any execution on this machine. Section 24.2 will therefore present in detail the SCI hardware of the SMiLE cluster and its accompanying hardware monitoring system. Section 24.3 then provides the background necessary to understand *codex* in general, while Section 24.4 explains in detail how this approach can be mapped onto the SMiLE architecture. A short description of related work follows in Section 24.5, leading to the conclusion in Section 24.6.

24.2 The Monitoring Approach for the SMiLE PC Cluster

The SMiLE project at LRR-TUM [4] attempts to leverage SCI technology to set up, and provide software for, a cost-effective PC cluster to be used as a high-performance parallel processing platform. One major research goal is to develop methods and tools for the efficient use of DSM systems with NUMA characteristics. In order to support this endeavor, we have developed our own custom PCI/SCI adapter card which allows the attachment of an SCI hardware monitor.

The PCI/SCI adapter, described in detail in [2], serves as the basis for the SMiLE PC cluster by bridging the PC's I/O bus to the SCI interconnection network. The PCI/SCI adapter intercepts processor-memory operations on the PCI local bus, generates packets for remote SCI nodes and forwards them to the SCI network. Vice versa, incoming packets arriving from the SCI network are transmitted to the neighboring SCI node via the output link, or

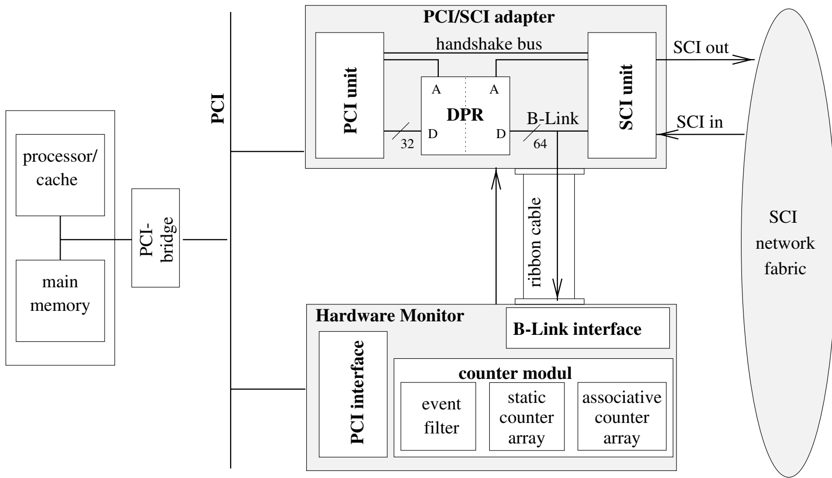


Fig. 24.1. The SMiLE SCI node architecture: a PCI/SCI adapter and the hardware monitor card installed in a PC

are translated into PCI transactions for the local node. SCI address spaces of the bridge can be mapped into local PCI addresses, allowing read or write accesses to any of the mapped areas. The PCI/SCI adapter is responsible for the address translations and the request/response packet generation.

As shown in Figure 24.1, the PCI/SCI adapter is divided into three logical parts: the PCI unit, the Dual-Ported RAM (DPR), and the SCI unit.

The PCI unit forms the interface to the PCI bus of the local PC host. Memory transactions of the local processor to the PCI bus referencing a 64-MByte address window within the physical address space are intercepted and then translated into SCI transactions. The packets to be sent from the local SCI node to remote memory are buffered within the DPR. The SCI unit in turn interfaces to the SCI network and performs the SCI protocol processing for packets in both directions. Here, the B-Link, a 64 bit-wide synchronous bus connecting the SCI unit and the DPR, serves as the carrier of all incoming and outgoing packets to and from the SCI interface.

The SMiLE SCI hardware monitor as part of the event-driven hybrid monitoring approach is attached to the PCI/SCI adapter as an additional PCI card as shown in Figure 24.1. The B-Link is the central point on which all remote memory traffic can be monitored. The information that can be recorded from the B-Link includes the transaction command, the target and the source node IDs, the node-internal address offset, and the data.

The flexible architecture of the hardware monitor allows the programmer to utilize it in two working modes for performance analysis: the *dynamic* and the *static* mode. *The dynamic working mode* (Figure 24.2) is suitable for delivering detailed information to tools for performance evaluation and tuning. In order to be able to record all data of interest with only limited

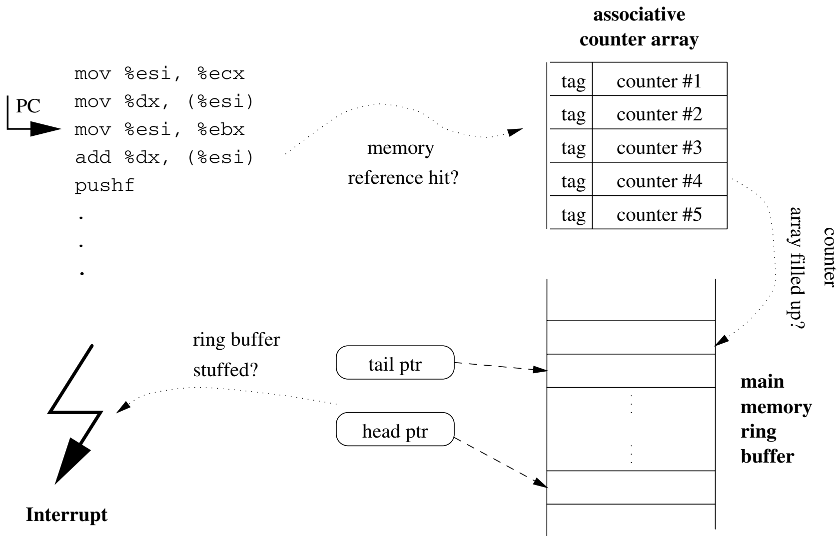


Fig. 24.2. The hardware monitor's dynamic mode working principle

hardware resources, the monitor exploits the spatial and temporal locality of data and instruction accesses in a way similar to cache memories in high-performance computer systems. Memory access events detected through this mechanism are then counted in a register file which is implemented as a content-addressable counter array managing a small working set of the most recently referenced memory regions. The remote memory accesses are detected by monitoring the B-Link. If a memory reference matches a tag in the counter array, the associated counter is incremented. If no reference is found, a new counter-tag pair is allocated and initialized to 1. If no more space is available within the counter array, first counters for neighboring address areas are merged or a counter-tag pair is flushed to a larger ring buffer in main memory. This buffer is supposed to be emptied by some performance evaluation tool in a cyclic fashion. In the case of a ring buffer overflow, a signal is sent to the software process urging for the retrieval of the ring buffer's data.

As the amount of flushing and re-allocating counter-tag pairs should be reduced, it makes sense to integrate the strategy of adapting counter coverage into the cache replacement mechanism. Under the prerequisite that counter tags can name not only single addresses but also contiguous memory areas for which the counter is to register the accesses, a simple least-recently-used (LRU) replacement algorithm can be adapted to this special task. The maximum address range, however, has to be predefined by the user. The method is called Dynamic Coverage LRU and is shown in Figure 24.3.

A detailed description and the rationale of the hardware monitor's *dynamic* working mode is given in [5]. As shown in that paper, a size of 16

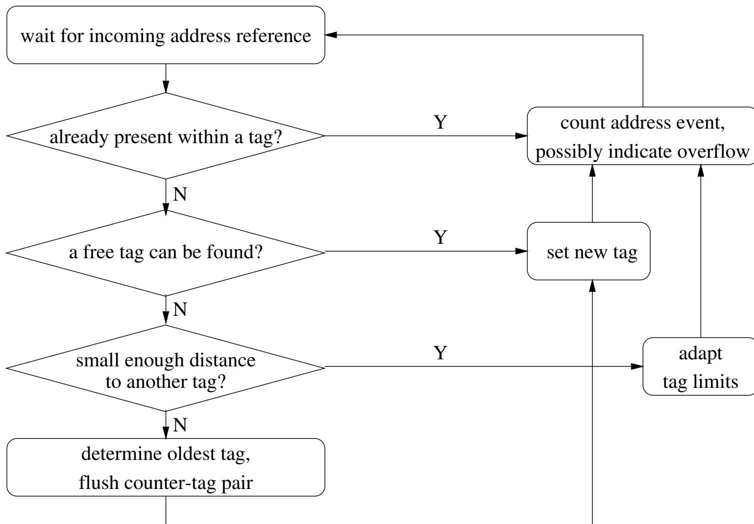


Fig. 24.3. The Dynamic Coverage LRU mechanism

counter-tag pairs is sufficient for the associative counter array in typical applications.

The *static working mode* allows users to explicitly program the hardware for event triggering and action processing on SCI regions. Figure 24.4 shows a simplified view of the hardware structure used to realize that feature.

The *event filter* comprises a *page table* and the *event station*. In combination, both implement the ability to monitor memory regions and particular transactions upon them. The page table contains the page descriptors, uniquely describing the pages within the SCI address space which are to be monitored. A page table's descriptor is formed by the SCI node number, and the page frame address within the node's address space, together with some state information for that page.

The *event station* specifies the exact events on which the monitor triggers. An entry within this hardware structure points to a page descriptor within the page table. The bottom and top address fields specify the address range to be monitored within the indexed page, while the transaction type can be used to restrict the SCI transactions to be registered. The transaction type consists of several flags for the events that can be monitored (read, write, and lock transaction, request and response transaction, incoming or outgoing packets, *codex* mode).

During performance analysis, each counter within the static counter array is able to act in three possible ways on the events described in the event station. While two of these ways are used to enable and disable the counter, the third way triggers the actual event counting. Enabling and disabling a counter temporarily is useful for hybrid monitoring approaches in which, e.g.,

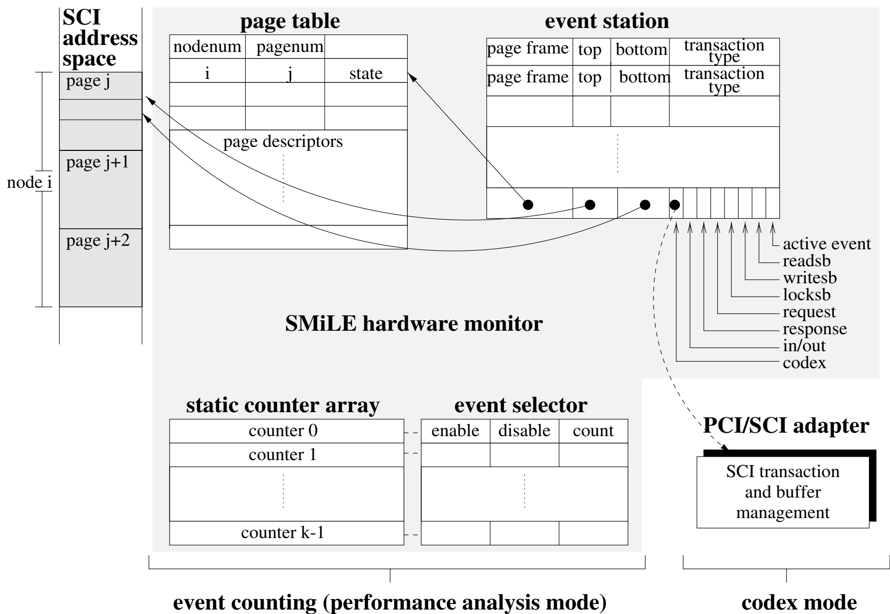


Fig. 24.4. The monitor's static mode hardware structures

only memory accesses during certain function invocations are to be monitored. An easy instrumentation of the function entry and the exit, generating events to switch on and off the respective counter, serves this purpose.

In order to support the controlled deterministic execution approach *codex* which will be explained shortly, the hardware monitor can be reconfigured. Instead of routing the recognized events to the counters, a set *codex* mode flag in the event station forces this signal to be provided to the transaction processing on the PCI/SCI adapter card. This particular way of using the monitor opens up new possibilities for debugging and testing fine-grained parallel programs on hardware-supported DSM architectures.

In addition to the hardware structures described, a number of registers and logic is necessary for configuration and control of the monitor. A physical implementation of the hardware monitor is currently under development.

24.3 The Controlled Deterministic Execution Approach (*CODEX*)

Controlled deterministic execution provides a method to overcome problems arising during a very important phase of software development of parallel programs, the phase of testing and debugging. Here, the degrees of freedom in the order of accesses to common resources may result in undesirable non-deterministic behavior. In the case of SCI, common resources are memory

regions of the common address space. With *codex*, the nondeterminism can be eliminated as far as it is necessary.

Codex is an independent approach to provide controlled execution for a couple of parallel programming models and environments. The independence is guaranteed by an abstract, object-oriented modeling method. This method is called POEM (Parallel Object Execution Model). Through its object-oriented nature, it is possible to adapt it to various communication models. The basic classes are called *Execution Objects* (EOs) and *Common Objects* (COs). EOs represent all kinds of threads of execution, i.e. processes, threads, or tasks. The media for the exchange of data, shared memory in case of SCI, is represented by COs. Accesses to COs, that represent communication operations, are reduced to two elementary operations: *read* and *modify*.

Accesses to COs by EOs are formulated using these elementary operations. Modify and read are utilized to construct the access modes to shared resources actually occurring in an application. While modify is regarded as a nonblocking operation, read has to be blocking by default. An example of a POEM is shown in Figure 24.5. A detailed presentation can be found in [3].

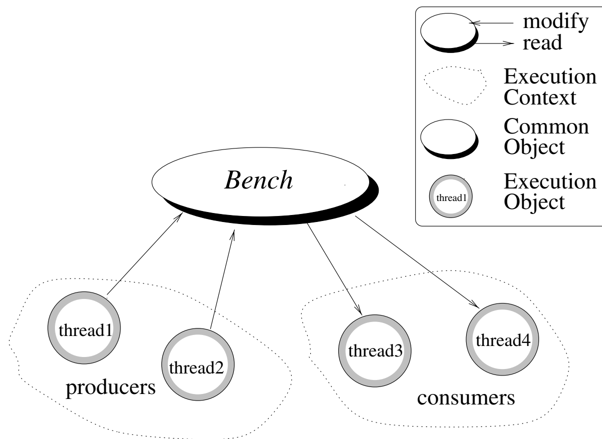


Fig. 24.5. POEM representation of a producer-consumer example

To enhance the representation and to provide scalability of POEM, we introduce *contexts*. A context is a group of objects of the same type. An *Execution Context* is a bag of EOs, while a *Communication Context* comprises a set of COs. Contexts behave like single objects. Therefore it is possible to hide a group of threads in a single context, thus increasing transparency. In Figure 24.5, we can see two contexts indicated by dotted lines: one for the producers and one for the consumers of the example. Figure 24.6 shows the reduced presentation as contexts.

For the specification of the communication behavior that needs to be enforced, some description technique is required. The description technique in our case is a context-free language that is called *Control Patterns* (CPs). It is based on POEM and allows the user to specify access sequences to COs. Unlike previous work [11], which uses a complete sequence of all relevant accesses, our first goal with CPs is to restrict the description to selected COs. Thus, we avoid long and tedious descriptions. Second, we reflect the fact of periodic repetitions of accesses to COs in our language. A pattern is exactly the description of periodic repetitions. Table 24.1 presents an overview of available operations to describe CPs. While the basic operations are similar to regular expressions, extended operations are especially suitable for the description of cyclic accesses.

Basic operations:		*Extended operations:*	
$a \succ a'$	– a has to be fulfilled before a'	$a!$	– a has the highest priority at the CO as long as it does not access another CO or it is blocked respectively.
a^x	– a must be fulfilled exactly x times	$a\#$	– a has the highest priority at the CO as long as it is not blocked.
a^{+x}	– a has to be fulfilled at least once but at most x times, $x \in N \cup \epsilon$		

Table 24.1. Operations of Control Patterns

The *codex* approach enforces the parallel program's execution according to such CPs.

Figure 24.6 shows the previous example enhanced with a simple CP notation which illustrates the use of contexts in the pattern expression. The notation guarantees alternate accesses of threads to the CO *Bench* from different logical thread groups, the *producers* and the *consumers*. The expression has to be interpreted as follows: before a consumer is allowed to access *Bench*, a producer has to modify it, and before the producer may put the next item on the bench, one item has to be consumed, due to the '+' operator. Thus, there is at most one item on the bench during the whole execution.

In a practical implementation, CPs correspond to restricting or enabling read and modify operations for the execution in a particular order. In essence, this requires explicit control over communication at the receiver's end of a connection.

A remaining question refers to the generation of a POEM and the derivation of CPs. In general, there are four possible sources for a POEM: specifications of program behavior, static analysis, event traces, and interactive composition by users. Currently, the description of CPs is carried out manually and is also supported by a graphical user interface. After the decla-

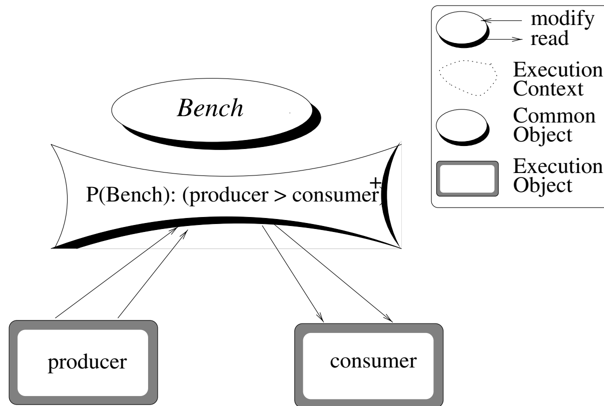


Fig. 24.6. The producer-consumer example with a Control Pattern

ration, all necessary information, the control information in particular, is delivered to the system controlling the execution.

A more detailed discussion of CPs and their use can be found in [10]. After this presentation of the idea of *codex*, the next section encompasses the adaptation of POEM to the runtime environment of the SMiLE cluster and its communication paradigm. With the adaptation we will be able to formulate Control Patterns to test or debug a variety of different access sequences to selected memory regions by using the hardware monitoring facilities.

24.4 Controlling Execution with SMiLE

24.4.1 Mapping POEM to the SMiLE Architecture

POEM's abstract model objects must have a counterpart in the SMiLE runtime environment. EOs map easily and naturally represent any task running on the SMiLE architecture. Depending on the programming model, these can be whole processes or single threads. COs on the other hand refer to either single memory cells or complete memory regions, which again will depend on the programming model of the application. Referring to a whole region is useful if a programmer thinks in terms of larger data structures instead of single memory cells. Accesses can be mapped in a straightforward way: SCI remote read transactions, e.g., the 16-byte selected-byte read (*readsb*), are an exact analogy to POEMs *read* operations. Similarly, SCI remote write transactions like the *writesb* and SCI remote read-modify-write transactions like *locksb* can be identified as POEM's *modify* operations. Table 24.2 summarizes these relationships.

POEM entity	SMiLE component
execution object	process thread
common object	memory cell memory region
read access	SCI remote read transaction (readsb)
modify access	SCI remote write transaction (writesb) SCI remote lock transaction (locksb)

Table 24.2. Mapping of POEM model objects to components or activities of the SMiLE architecture.

24.4.2 Controlling Execution on SMiLE

A challenging task for the use of *codex* on SMiLE is the part of controlling the parallel application at runtime or, in POEM terminology, of implementing an on-line Control Pattern method on this kind of architecture. The CP for this POEM imposes a particular total order on communication operations referring to a CO. It is therefore our task to enforce a particular execution order of SCI remote memory operations on common memory regions. Basically, this can be done at two locations:

1. At the active side of the communication, i.e., at the site of the EO. On SMiLE, this means controlling the processor triggering a remote memory access.
2. At the passive side of the communication, i.e., at the site of the CO. On SMiLE, this means controlling the execution of an SCI transaction at the receiver node.

While the first method seems more natural, it is actually quite difficult to be implemented. From the programmer's and the processor's point of view, memory operations via SCI look identical to regular memory operations. The only difference is in the address, since SCI transactions are started whenever a particular address window is accessed. Filtering out remote memory operations here would consist of detecting an address space during runtime, a task usually performed by the memory management unit (MMU) of the processor. An access to a certain window would trigger an exception and result in starting an appropriate interrupt service routine (ISR), which would have to perform the appropriate actions for realizing *codex*.

The SMiLE approach to controlled deterministic execution combines this method with the second solution. As the CP governs accesses *to* the CO, it is easier to be implemented on the SMiLE hardware. The SCI transactions are split into a *request* and a *response* sub-action, thus the receiving side of the communication provides an ideal location for the necessary intrusion into

communication behavior. Controlling execution on SMiLE thus comprises the following three steps:

1. Detection of accesses to COs. The SMiLE hardware monitor on the destination node and the MMU on the local node serve as detectors of accesses to COs by checking the address of each transaction. While the monitor signals this by a simple trigger line to the PCI/SCI adapter, the previously mentioned coupling between the event filter and the PCI/SCI transaction processing on the adapter card, the MMU raises an exception on the local processor. The reasons for using the MMU are twofold. First, it is the only means for detecting accesses if COs and EOs are located on the same node in the system. Additionally, it avoids stalling the local processor during a remote read operation, thus avoids deadlocks in servicing concurrent CP executions.
2. Saving accesses in a transaction pool. The signal causes the slightly modified PCI/SCI card not to execute the incoming transaction request, thus using the DPR buffers for intermediate storage of the SCI request.
3. Execution of accesses according to the predefined CP. From the transaction pool, SCI memory operations can then be executed under software control, the appropriate response sub-actions can be generated and sent back to the originator.

The first two steps can be performed with the existing SMiLE hardware while the third activity is part of the software package implementing *codex* for a particular programming model. The SMiLE monitor is used to "snoop" SCI transactions on the B-Link connecting the DPR and the Link Controller on the PCI/SCI adapter. The event filter is thus used to detect any access to a CO on the node placed on the remote side of a memory transaction.

In detail, the mechanism works as follows: the address recognition in the monitor event filter generates a signal that is routed back to the PCI/SCI adapter card via the flat ribbon cable. The PCI/SCI adapter is modified such that it does not execute an incoming transaction that has been placed in the DPR by the SCI unit of the card. Progress of the incoming SCI transaction that would normally be translated into appropriate PCI transactions, is stopped here. Fortunately, the microcode sequencer controlling the PCI side of the adapter can be programmed easily to perform this task. This also demonstrates the flexibility of the microprogrammed implementation approach for the PCI/SCI adapter as described in Chapter 4.

Only by directly routing the detection signal into the PCI/SCI adapter it is possible to meet the tight timing requirements of this approach, as the latency after the arrival of an SCI transaction until its access to the node's PCI bus is only about 1.2 μs [1]. This naturally leaves no room for operating system-based signals and software solutions up to this point.

The third step, however, is finally up to the software. Software interpretation of the readable DPR buffers as well as simply letting the adapter execute the transactions by selectively re-enabling them are two possible ways to do this.

24.4.3 A Framework for an Implementation of *CODEX* for Fine-Grained DSM Execution

The previously mentioned extensions to the SMiLE SCI hardware allow us to build a software system that realizes controlled deterministic execution. This section will give a brief outline of a facility that forms the basis of such systems. The following steps have to be implemented:

1. Generation of a CP according to the formalism described in Section 24.3. This is a task beyond the scope of this paper and is covered by the user interface to *codex*.
2. Establishing a mapping between COs and shared memory regions as well as a mapping between EOs and processes or threads on their nodes for the given application. This again depends massively on the programming model and is discussed no further here.
3. Use of the SMiLE hardware to enforce CPs on COs, i.e., memory accesses to particular regions. This comprises:
   – Presetting the monitor event filter to the address regions belonging to the COs.
   – Setting the software-implemented state machines that represent the CPs to their initial states.
   – Setting up the appropriate EO-side CP structures at the nodes of the EOs. This includes setting read protection through the MMU.
   – Execution of each EO on each node can thus be started. The *codex* functionality is automatically invoked at each read access to a CO by means of a page fault handler.

Figure 24.7 displays how the different parts of the system work together. On the node running an EO, MMU read protection is enabled for those memory areas containing COs, while store operations are forwarded via SCI to the remote node hosting the COs. An attempted read of a CO therefore causes the *codex* system to be invoked. As data is needed for continuing the EO execution and this data has to be consistent with the previously described CP, the node hosting the COs is requested to perform whichever transaction in its request buffers conforms to the CP. Finally, result data is returned to the EO node, allowing it to restart the original read operation.

For COs and EOs being on the same node, the identical mechanism can be implemented by merely using the MMU for read and write protection of

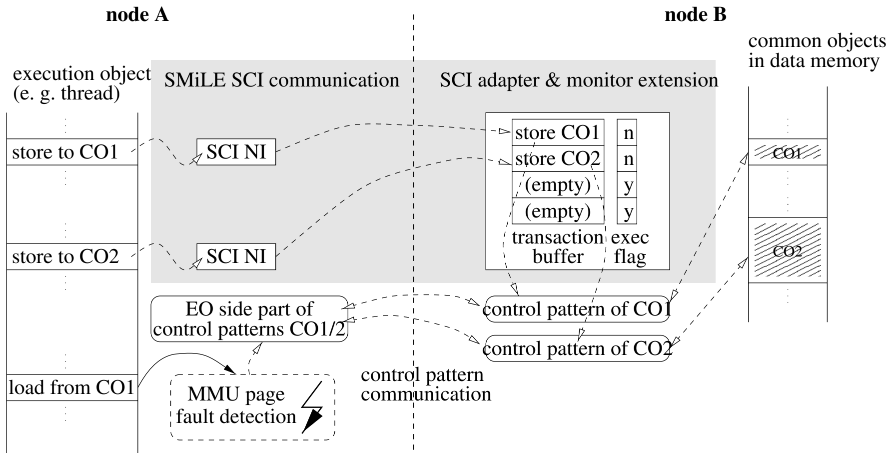


Fig. 24.7. Technical implementation of the *codex* functionality on the SMiLE SCI hardware

COs. While for a write page fault, accesses are queued in an area comparable to the SCI request buffers, the read protection invokes the same routines as in the previously explained distributed case.

Flushing the buffer, i.e., starting the processing of the buffered transactions due to a CP, is necessary before a buffer overflow might occur. Buffer space on the PCI/SCI adapter is naturally limited, yet it is hard to foresee generally by statical analysis how many write transactions will actually occur until the buffers in the network interface (NI) can be flushed. Thus, *codex* requires a conservative buffer management, implemented through a simple round-robin execution approach. For a maximum number of N buffers reserved for controlled deterministic execution[1], we require that each node hosting EOs must not issue more than N write transactions before the *codex* management is invoked again. Instrumentation of the application is a safe way to do this: a call graph analysis reveals which machine operations in which basic blocks are candidates for CO accesses. Should a walk through the graph indicate that the amount of N write operations to COs will potentially be exceeded, additional calls to the CP management can be inserted. These calls as well as the page fault handler-invoked CP software cause non-serviced write transactions to be flushed out of the NI card buffers into the same CO-sided queues that are also in use for single-node shared memory. After that, the next node, or rather the next EO, can be selected to be run the same way.

[1] It is advisable to use only a fraction of the actual number of buffers for *codex* as, e.g., management communication of the *codex* system should go undisturbed.

24.5 Related Work

Only a few DSM monitoring projects so far focused on providing hardware support for hardware-based distributed shared memory. The performance monitor for the Princeton SHRIMP multicomputer [8] is one example. It can be configured to run as a trace monitor, using a high amount of local memory to store local traces without probe effect, or as a multidimensional histogram monitor. For the CC-NUMA FLASH multiprocessor system [7], the hardware-implemented cache coherence mechanism is complemented by components for monitoring fine-grained performance data (number and duration of misses, invalidations, etc.) [9]. Some multiprocessor systems exploit information gathered by hardware counters incorporated within modern CPU chips (DEC Alpha 21164, Intel Pentium processors, MIPS R10000) for performance analysis [12]. Yet, none of these systems actually attempts to influence program behavior in addition to performance measurement.

24.6 Conclusion

This chapter introduced the rationale, concept, and implementation of an infrastructure enabling performance evaluation and controlled deterministic execution on hardware-supported DSM architectures. The SMiLE system [4] served as the example for such a system in which PCs are clustered by a modern low-latency and high-speed interconnect like SCI.

While the fine-grained nature of communication on this class of parallel and distributed computers provides the advantages of a finer grain of computational activities, it aggravates the development of efficient parallel programs on these machines. For both performance analysis and debugging, observability of communication is one of the key elements of gaining insight into the behavior of the program under test.

For this task, the SMiLE hardware monitor card offers the only possibility to detect the sizes, the points in time, or the destination addresses of the underlying SCI memory transactions. For performance analyses, this information can be logged in the nodes' main memories in an efficient manner w. r. t. both time and space requirements. A second task of a monitor usually consists of influencing the behavior of a program run. Again, the combination of the SMiLE monitor card and the SMiLE PCI/SCI adapter provides the means for this by selectively detecting and stalling incoming SCI transactions. A software part of a hybrid monitoring system is now free to re-enable these transactions explicitly and thus introduce explicit control into the memory-oriented communication provided by the SMiLE system architecture. We presented the *codex* system as an example of the power of this approach. Controlled deterministic execution that is normally only considered feasible for message-passing execution is made possible for a distributed

shared memory system model. By this, the SMiLE monitor is a unique tool both for performance debugging as well as for program testing.

The SMiLE hardware monitor is currently being implemented as an FPGA-based PCI card. The VHDL-based design flow helps to keep the design and some architectural parameters such as the numbers of counters as flexible as possible. In [5] we have shown that its cache-like architecture, exploiting temporal and spatial locality of remote memory accesses, enables us to build a resource-saving hardware even for heavy performance monitoring, while avoiding most of the undesirable probe effect of software instrumentation.

References

1. G. Acher, H. Hellwagner, W. Karl, and M. Leberecht. A PCI-SCI Bridge for Building a PC Cluster with Distributed Shared Memory. In *Proc. 6th International Workshop on SCI-based High-Performance Low-Cost Computing*, pages 1–8, SCIzzL, Santa Clara, CA, Sept. 1996.
2. G. Acher, W. Karl, and M. Leberecht. PCI-SCI Protocol Translations: Applying Microprogrammable Concepts to FPGA. In R. Hartenstein and A. Keevallik, editors, *Proc. 8th International Workshop on Field Programmable Logic and Applications (FPL'98)*, volume 1482 of *Lecture Notes in Computer Science*, pages 99–108, Springer Verlag, Aug. 1998.
3. M. Frey and M. Oberhuber. Testing and Debugging Parallel and Distributed Programs with Temporal Logic Specifications. In *Proc. 2nd Workshop on Parallel and Distributed Software Engineering*, pages 62–72, IEEE Computer Society, May 1997.
4. H. Hellwagner, W. Karl, and M. Leberecht. Enabling a PC Cluster for High-Performance Computing. *SPEEDUP Journal*, 11(1), June 1997.
5. R. Hockauf, W. Karl, M. Leberecht, M. Oberhuber, and M. Wagner. Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In D. Pritchard and J. Reeve, editors, *Proc. 4th International Euro-Par Conference (Euro-Par'98)*, volume 1470 of *Lecture Notes in Computer Science*, Springer Verlag, Sept. 1998.
6. W. Karl and M. Leberecht. Ein Monitorkonzept für Systeme mit verteiltem gemeinsamen Speicher. In R. Hoffmann, B. Klauer, C. Müller-Schloer, K. D. Reinartz, and H. C. Zeidler, editors, *ARCS'97: Architektur von Rechensystemen 1997. Vorträge der Workshops im Rahmen der 14.ITG/GI-Fachtagung*, Universität Rostock, pages 169–178, Sept. 1997.
7. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, pages 302–313, ACM, 1994.
8. M. Martonosi, D. W. Clark, and M. Mesarina. The SHRIMP Performance Monitor: Design and Applications. In *Proc. 1996 SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96)*, pages 61–69, ACM, May 1996.
9. M. Martonosi, D. Ofelt, and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. In *Proc. 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96)*, ACM, 1996.

10. M. Oberhuber, S. Rathmayer, and A. Bode. Tuning Parallel Programs with Computational Steering and Controlled Execution. In *Proc. HICSS-31*, pages 157–166, Jan. 1998.
11. K. Tai and R. Carver. Testing Distributed Programs. Chapter 33 of *Parallel and Distributed Computing Handbook*. McGraw Hill, New York, USA, 1996.
12. M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proc. Supercomputing (SC'96)*, 1996.

25. Monitoring SCI Clusters

Matthias Maier-Stahel[1], Roger Butenuth[2], Hans-Ulrich Heiss[3]

[1] University of Paderborn, Germany
email: stahlie@uni-paderborn.de
[2] University of Paderborn, Germany
email: butenuth@uni-paderborn.de
http://www.uni-paderborn.de/cs/butenuth.html
[3] University of Paderborn, Germany
email: heiss@uni-paderborn.de
http://www.uni-paderborn.de/cs/heiss.html

25.1 Motivation

The more complex a computer system is, the more important it is to get relevant information about its operational state. In a multi-user environment, there is usually an operator or administrator who is responsible for smooth operation. He or she needs to be aware of any abnormal behavior, e.g. node failure, overload situations, deadlocks, bottlenecks or other situations related to availability and performance. To that end, a console is used to inform the operator about the state and the behavior of the machine at one single place.

In a system with a single copy of the operating system (e.g. an SMP system), such a console is a standard feature. An SCI cluster, however, is more complicated. Although it provides physically shared memory and can therefore be considered a NUMA multiprocessor, its "look and feel" to the user is rather a collection of autonomous nodes each running a complete and independent local operating system. Redirecting console output of the individual nodes to a central terminal is possible but not sufficient, since a node usually simply crashes without sending a message in advance. An operator of an SCI cluster would have to probe the nodes to make sure that all of them are up and running.

In addition to the operational states of the nodes, the operator also wants more detailed information about the utilization and performance of the system, since any anomaly in system behavior may indicate a situation that needs human intervention. A component that provides this kind of information is usually called a *monitor*. A monitor observes the system by sampling relevant system measures, such as utilization, throughput, and other quantities and makes these measurements available for on-line or off-line analysis. In a multi-programming environment, it should be possible to attribute the measured quantities to the individual programs, offering some insight into their behavior. By providing this functionality, a monitor can help the programmer debug the parallel program or reveal design flaws leading to poor performance. The monitor provides a global bird's-eye view of the system, which is usually not available in a distributed system.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 433-441, 1999
© Springer-Verlag Berlin Heidelberg 1999

In the following, we present a monitoring tool that has been developed for SCI cluster computers. Section 25.2 gives an overview of its general structure. Sections 25.3 – 25.5 describe the major components and the way they interact. A short conclusion in Section 25.6 closes this chapter.

25.2 General Architecture

The architecture of a monitoring tool for a cluster machine derives quite naturally from its distributed architecture. On each node, a local monitoring agent is needed that samples data locally and provides it to a central master component for combination and further processing. The communication protocol between the agents and the central master is one of the major design issues, which will be discussed shortly. The data collected by the master is made available to other components. One of these clients is the *visualizer* that presents the performance data to the human user in an easily comprehensible way and provides a global view on the system's state. Another component fed by monitoring data is the *alarm tool*, which can be used in addition or instead of the visualization component. In its first and basic version, it simply calls for the user's attention acoustically and visually if one of the nodes crashes. It can be enhanced by more flexible trigger conditions defined by the user. Another possible user of monitor data could be a *cluster resource manager* responsible for optimizing the utilization of the overall machine, e.g. CCS (see Chapter 26).

The monitor and its components are implemented mainly in C, while the visualizer is based on Tcl/Tk scripts and the X window system [7]. The whole package is available for the Intel x86 processor family and also for Sun SPARC processors. The operating systems supported are Linux and Solaris. The monitor also works in a heterogeneous cluster, since it uses the Sun XDR presentation layer software [1] to perform data conversion for any cross-platform communication. All communication within the monitor and to the outside world is based on standard TCP/UDP/IP protocols. Therefore, any communication medium can be used, as long as it is fast enough and IP multicast is available. In the current implementation, Fast Ethernet is employed. Figure 25.1 shows the layers of the resulting monitoring system.

25.3 Monitor Agents

The main purpose of the monitor is to assist the system administrator in running the cluster. Although it is also helpful for debugging, this is not the major goal. Monitoring for operating systems is targeting at rather coarse time scales. It is therefore sufficient to get averaged data using a time driven approach. Triggered by the master periodically (e.g. each second by default),

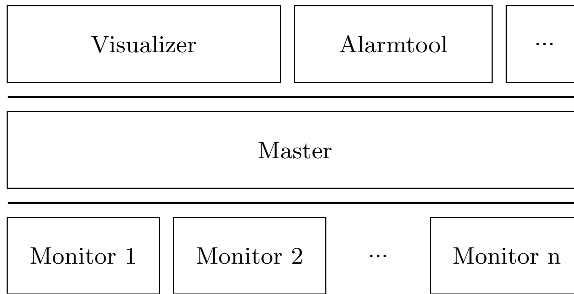


Fig. 25.1. Monitoring system

the local monitor agents respond by sending the status and measurement data back to the master.

The relevant data the monitor provides can be divided into static data and dynamic data. The static data is collected when the monitor is started and does not change during the runtime of a system. Dynamic data results from measurements taken over time, by calling appropriate commands specific for each operating system.

1. Static data:
   – host name
   – IP address of the host
   – version of the operating system
   – type of processor
   – number of processors
   – amount of main memory
   – amount of swap space
   – amount of disk space of the root partition
   – SCI ID of the node
2. Dynamic data:
   – average load of the processors
   – amount of main memory used
   – amount of swap space used
   – amount of disk space used (root partition)
   – SCI error rate (Linux only)
   – existence of two processes needed by the cluster management software (CCS)

By encapsulating all data specific actions into separate procedures the set of data monitored can be expanded easily.

Generally, the collection of the local data at the central site can be initiated either by the local agents or by the central master. As with all software monitors, the observer influences and distorts the observed system. Especially

in a parallel computer where the processes of a parallel program may communicate synchronously, care should be taken not to disturb the programs too much. Each activation of a local monitoring agent causes an interruption of the local application process and results in a delay. If the agents take the initiative to update the measurements, then, in the worst case, all agents will become active one after the other and the delays at the nodes will mutually aggravate each other, resulting in poor performance of the application. To prevent this situation and to minimize the impact of monitoring on program behavior, we use a polling approach where the master requests the updates from the local agents in a multicast operation. The multicast enforces the data collection in a rather synchronous way so that all nodes are delayed roughly at the same time.

25.4 Master

The monitor has been designed to be used in cooperation with the CCS Resource Management Software (see Chapter 26). Because CCS manages the cluster computer it holds the necessary information about its architecture, such as number and names of nodes, interconnection topology, and makes it available to the monitor master in a file. Having read this data, the master builds a data structure for each node containing its static and dynamic data.

After this initialization, the master periodically sends multicast messages to the cluster nodes requesting an update of measured quantities. The monitor agents respond by sending the requested information back to the master. Multicast has been used to achieve a high degree of synchronization in message delivery at a low overhead. Compared to broadcast it is more flexible, since it allows to distinguish different clusters in the same subnet. Multicast is based on the User Datagram Protocol (UDP), which is the connectionless alternative to TCP in the Internet protocol suite [2]. The smaller overhead of UDP results from the fact that UDP does not care about packet loss. Although packet delivery at the hardware level is reasonably reliable within a cluster machine, packets may get lost due to overloaded nodes and buffer overflow. Dealing with packet loss by retransmission would be too expensive and inappropriate in this case. The loss of a packet only means the loss of one sample of a single node, which is simply replaced by the sample acquired one time step before. The monitor agents are also using UDP to return their measurements.

UDP messages are numbered so that the answer of a node can be matched to the corresponding request of the master. This enables the master to recognize whether an answer is out of date. Outdated answers are simply dropped but counted. If there are k (default: k=5) answers missing, the master decides that the node has crashed and marks its data structure accordingly.

Besides the measurements from the nodes, the master receives information about the partitioning of the cluster by CCS. Each time the allocation of

cluster partitions to the parallel programs changes, the master is notified. The data collected by the master is made available to other components through a small interface library. There are currently two tools using monitoring data: the visualizing tool and the alarm tool. The general communication structure and the protocols used are shown in Figure 25.2.

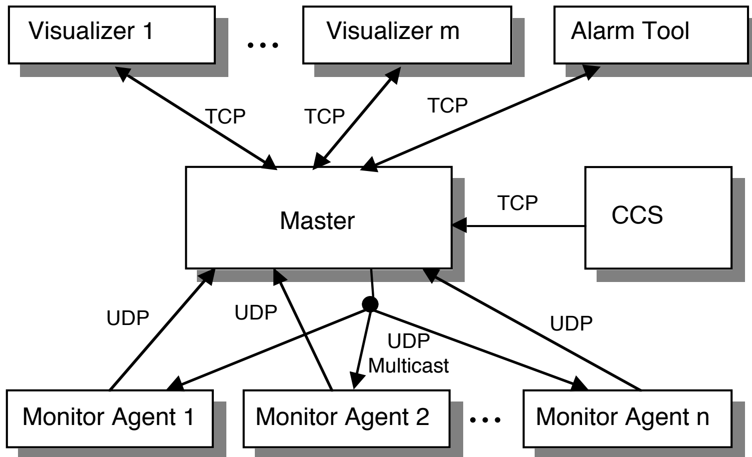


Fig. 25.2. Communication structure

25.5 Visualizer

The visualizer is a graphical front-end which displays the data collected by the master in order to provide a global and easily comprehensible view on the system and its state. It is intended to be used by the system administrator but also by users running parallel programs.

At start-up the visualizer first connects to the master and demands detailed information about the cluster as a whole. This is basically the same information that the master obtained from the CCS, i.e. the number of nodes and their names, and the topology of the interconnect. In a second step, the static data of the individual nodes is requested. Both types of information are kept in local data structures of the visualizer. The dynamic data is requested periodically, with a rate adjustable by the user. The default value is five seconds. The master replies to these requests by sending the whole list of node specific data as well as the CCS information. To draw diagrams that visualize the variation of the data over time, the actual values of the most recent τ measurements are stored.

The communication between the visualizer and the master is based on the reliable TCP (Transmission Control Protocol), which is preferable to UDP whenever its larger overhead is uncritical. Since there are only a few visualizers running at the same time, the master is able to service the corresponding TCP connections. To provide the whole picture for the user, the visualizer shows the data of each node separately. The visualizer is designed to support meshes and tori up to a dimension of three. Since there are currently only 2D meshes or tori in use, the layout of the visualization window corresponds to the cluster topology and shows the nodes arranged in a 2D matrix. For 3D topologies each 2D plane would be shown separately.

Each node is represented as a square icon with a bar chart indicating the quantitative data. The user can configure which parameter of the data set is shown by which bar and the assignment is indicated by a legend. The values displayed by the bars are percentages of their maximum values. To increase the readability for the human user, the bars also change their color from blue to red as the numbers increase. In each bar, a horizontal line indicates the average value, calculated as the arithmetic mean over all nodes. So it is easy to see whether a node is above or below the average and by how much. The bar chart representation is the result of a small empirical study that was carried out in cooperation with the visualization research group of the University of Paderborn. Among a handful of different graph types proposed for data representation, the bars turned out to have the best readability. Failed nodes are shown as crossed out icons. As mentioned in Section 25.3, the existence of critical system processes (used by CCS) is also monitored. In case of a failure of one of these processes, the node icon is crossed out by a yellow cross, while a total node failure is shown by a red cross.

The partitioning of the cluster to different parallel programs is visualized by different frame colors of the squares. Free nodes are shown in black. The colors of the partitions and the names of the occupying programs are depicted at the bottom of the window. Figure 25.3 shows a screen-shot of the visualizer window. It presents the state of a cluster which is connected as an 8×4 torus. Currently, there are three partitions in use, one consisting of 16 nodes placed at the bottom, eight nodes above them and two in the upper right corner. The scalability of the visualizer in terms of the number of nodes that can be shown with sufficient readability is naturally limited by the space available on a standard 17-inch or 19-inch display. However, since the size of the icons is adjustable, they can be made smaller to fit even hundreds of icons on a screen. The readability will clearly suffer, but the bar charts in connection with the color encoding are able to catch the viewer's attention, especially in situations with uneven load distribution, since even in small icons highly loaded nodes appear as red blocks, contrasting conspicuously to lightly loaded nodes shown in background color.

More detailed information about a node is available in separate windows. Each node is represented by such a window, which pops up whenever the

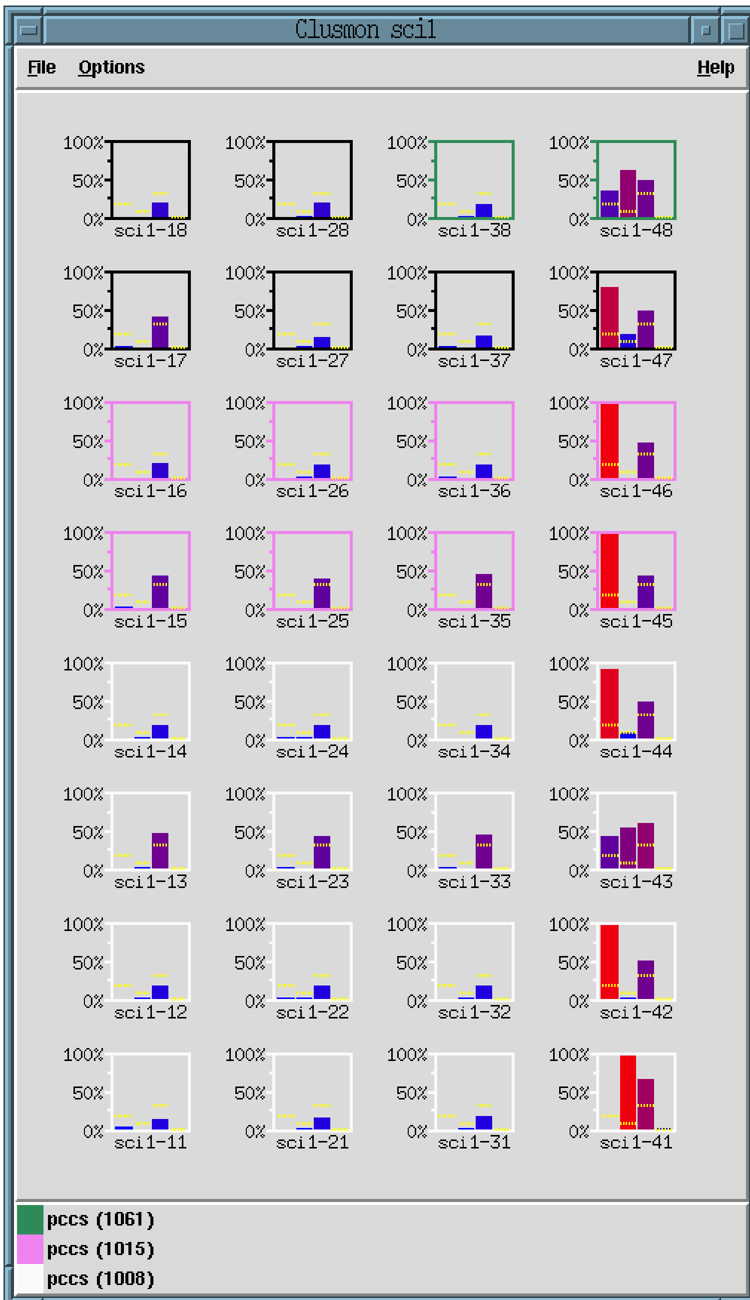


Fig. 25.3. Graphical front-end

corresponding icon is clicked by a mouse button. The window consists of two parts. The upper part shows a textual description of the node including the static and the dynamic data as numerical values as well as the data provided by CCS. The lower part of the window shows some curves displaying the values of selected measures over the last τ samples. Such a window is shown in Figure 25.4.

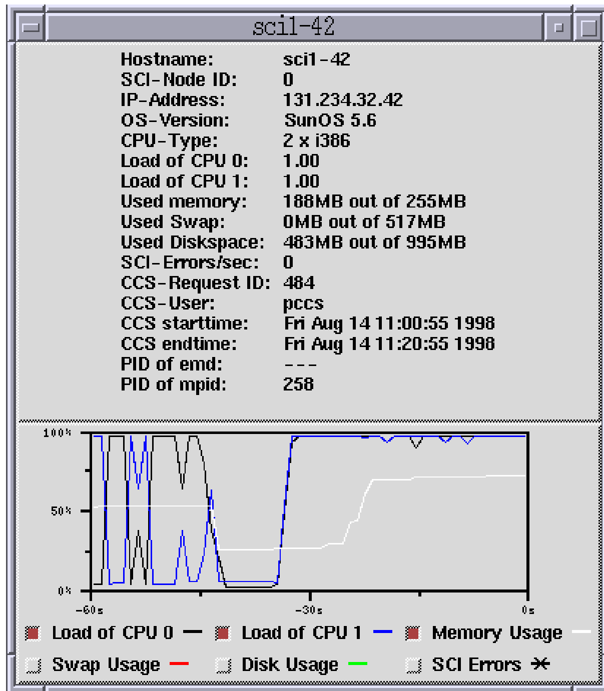


Fig. 25.4. Node-specific window

25.6 Conclusion

Computer clusters are difficult to manage if there is no single system image available. The monitoring tool is one step to fill this gap by providing a global view of the operational state of the cluster and its resource utilization. This information can also help the programmer in debugging and tuning. Its core consists of local monitoring agents and a central master process communicating in a master-worker fashion. A visualizer presents the data collected to the human user in a clear and well readable way. All components

are flexible and configurable to meet the users' needs. A small library makes the monitoring data available to other components to further improve the efficiency and ease of use of a cluster computer.

References

1. C. Brown. *Programmieren verteilter UNIX-Anwendungen.* Prentice Hall, 1994.
2. D. Comer. *Internetworking with TCP/IP.* Prentice Hall, 1988.
3. A. Langsford, J. D. Moffet. *Distributed Systems Measurement.* Addison-Wesley, 1993.
4. N. Luttenberger. *Monitoring von Multiprozessor- und Multicomputersystemen.* Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung der Universität Erlangen-Nürnberg, Band 22, Nummer 7, 1988.
5. M. Maier-Stahel. *Erfassung und Visualisierung des Systemzustands in einem Clusterrechner.* Universität Paderborn, 1998. http://www.uni-paderborn.de/fachbereich/AG/heiss/diplomarbeiten/visualisierung.html.
6. A. Nye. *Xlib Programming Manual.* O'Reilly & Associates, 1988.
7. J. K. Ousterhout. *Entwicklung grafischer Benutzungsschnittstellen für das X Window System.* Addison-Wesley, 1995.
8. S. A. Rago. *UNIX System V Network Programming.* Addison-Wesley, 1993.
9. M. Santifaller. *TCP/IP und ONC/NSF in Theorie und Praxis.* Addison-Wesley, 1993.
10. K. Waldschmidt (Hrsg.). *Parallelrechner: Architekturen - Systeme - Werkzeuge.* B. G. Teubner, 1995.

26. Multi-User System Management on SCI Clusters

Matthias Brune[1], Axel Keller[2], Alexander Reinefeld[1]

[1] Konrad-Zuse-Zentrum für Informationstechnik, Takustr. 7, D-14195 Berlin
email: {brune,ar}@zib.de
http://www.zib.de/

[2] Paderborn Center for Parallel Computing, D-33102 Paderborn
email: kel@upb.de
http://www.upb.de/pc2/

26.1 Introduction

The growing maturity of hardware and software components has tempted researchers to build very large SCI clusters with several hundred processors that are operated as high-performance compute servers in multi-user mode.

In this chapter, we present a resource management software for the user access and system administration of high-performance compute clusters named *Computing Center Software (CCS)*. It is in day-to-day use since 1992 on various parallel systems and has recently been adapted to the management of SCI clusters. CCS provides pluggable schedulers, optimal space partitioning for multiple users, reliable user access, and powerful tools for specifying resources and services by means of a specification language and a graphical user interface.

After a brief introduction in the remainder of this section, we describe the CCS system architecture and the characteristics of its resource description facilities.

26.1.1 Hardware Scenario

While CCS can be also used for accessing and controlling small heterogeneous SCI clusters like the one shown in Fig. 26.1a, it was primarily designed for managing large dedicated compute clusters that are operated in multi-user mode. Figure 26.1b depicts our 32-node SCI cluster. Its 2D torus topology is made up of four vertical and eight horizontal SCI rings. Each node is equipped with two Pentium II processors at the intersection points. Due to the different physical ring lengths, the vertical and horizontal rings exhibit different communication bandwidths of 400 and 500 MByte/s, respectively.

Our second system, shown in Figure 26.2, has a peak performance of 86 GFlop/s. It comprises 96 nodes, each with two 450 MHz Pentium II processors

[†] The work presented in this chapter was done while all three authors were at Paderborn Center for Parallel Computing, http://www.upb.de/pc2

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 443-461, 1999
© Springer-Verlag Berlin Heidelberg 1999

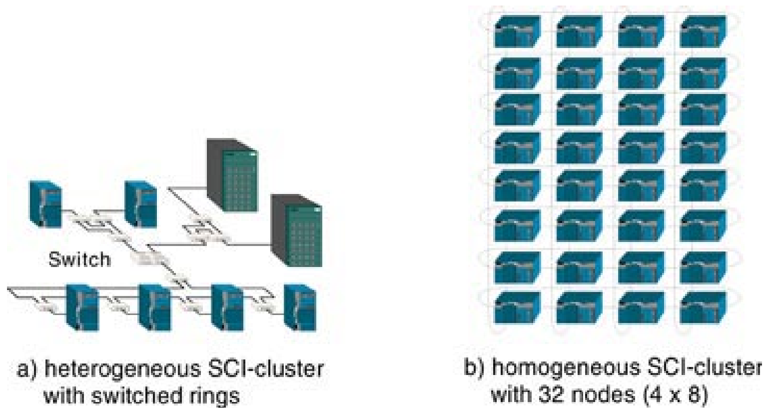


Fig. 26.1. SCI cluster configurations at PC^2

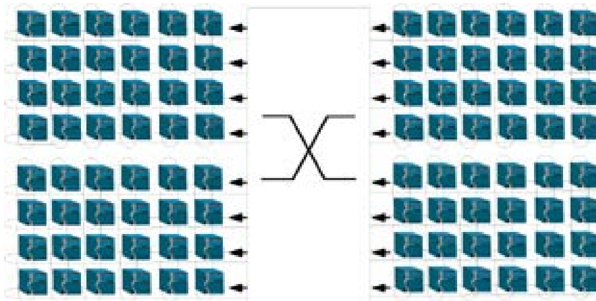


Fig. 26.2. Topology of a switched SCI cluster with 96 nodes (=192 processors)

and 512 MB main memory. The SCI rings are routed via a 16-way SCI switch. Again, this system is also operated under CCS.

26.1.2 Software Scenario

On the software side, our SCI clusters provide a full range of software services known from other high-performance computing environments. This includes a spectrum of compilers (Fortran77, Fortran90, C, C++), programming interfaces (PVM, MPI, Active Messages), a parallel debugger (TotalView) and a performance monitor (Vampir).

With different device drivers for Solaris, Linux and Windows NT it is possible to run multiple operating systems on different partitions at the same time. This is sometimes named *dynamic domains* concept. The dynamic repartitioning mechanisms in CCS are based on the low-level *ScaConf* software which allows to change partition sizes and to remove nodes in case of failure.

26.1.3 User Access and System Management

CCS has the same look-and-feel as traditional high-performance computer management systems. As a matter of fact, CCS originates from the transputer world, where massively parallel systems with 1024 processors had to be managed [18] by a single resource management software. The first CCS software release had the following features:

– concurrent user access to exclusively owned resources,
– coherent management of interactive and batch jobs,
– optimal system utilization by dynamical partitioning and scheduling,
– fault tolerance for remote user access via WANs.

We later added various pluggable scheduling strategies, such as *deadline scheduling* and schedulers for multi-site applications. For SCI clusters, optimal user partitioning over rings and switches have been included.

Thus, compared to workstation cluster management systems like LSF [16], Codine [10] or Condor [15], CCS is optimized for the efficient handling of parallel applications considering topology constraints. Due to its modular concept, the CCS software runs on many UNIX systems such as AIX, Linux, and Solaris.

In the following, we describe the architecture of CCS and present the resource and service description tools that are used by CCS to specify the available system components.

26.2 Architecture of CCS

26.2.1 Island Concept

In a computing site with several high-performance systems one might be tempted to operate all systems under the supervision of one central resource management system that has different backends for the various machines. On the one hand, this approach provides a coherent user and administrator interface to all machines, but on the other hand, it is inherently vulnerable to single points of failure. Moreover, the central scheduler—and other critical software modules—might cause a performance bottleneck. We therefore introduced the *Island Concept* [13], where each machine is managed by a separate instance of the CCS software (Fig. 26.3). A CCS island consists of six components:

– The *User Interface (UI)* offers X-window or ASCII access to the machine. It encapsulates the physical and technical characteristics and it provides a homogeneous access to single or multiple systems.
– The *Access Manager (AM)* manages the user interfaces and is responsible for authorization and accounting.

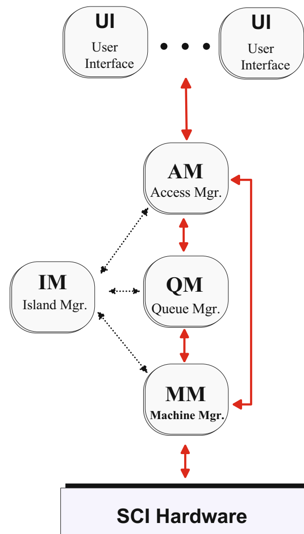


Fig. 26.3. Architecture of a CCS Island

- The *Queue Manager (QM)* schedules the user requests onto the machine.
- The *Machine Manager (MM)* provides an interface to the machine specific features like partitioning, scheduling, etc.
- The *Island Manager (IM)* provides name services and watchdog functions to keep the system in a stable condition.
- The *Operator Shell (OS)* is the main interface for system administrators to control CCS, e.g. by connecting to the system daemons (Fig. 26.4).

26.2.2 User Interface

The *User Interface (UI)* runs in a standard UNIX shell environment like tcsh. Common UNIX mechanisms for I/O re-direction, piping and shell scripts can be used. All job control signals (ctl-z, ctl-c, ...) are supported. The user shell accepts five commands:

- *ccsalloc* for allocating and/or reserving resources,
- *ccsrun* for starting jobs on previously reserved resources,
- *ccskill* for resetting or killing jobs and/or for releasing resources,
- *ccsbind* for re-connecting to a lost interactive application/session,
- *ccsinfo* for getting information on the job schedule, users, job status etc.

The *Access Manager (AM)* analyzes the user requests and is responsible for authentication, authorization and accounting (product of CPU-time and #nodes). CCS is able to handle project specific user management. Privileges

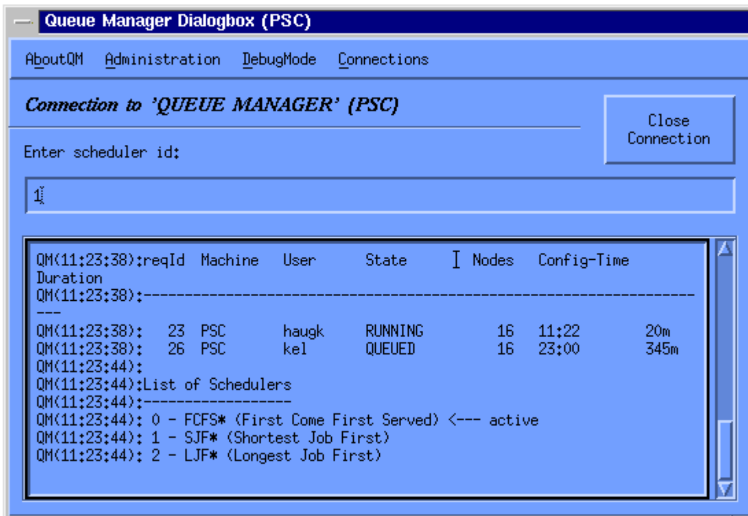


Fig. 26.4. The CCS operator shell

can be granted to either a whole project or to specific project members, for example:

– access rights (batch, interactive, the right to reserve resources),
– allowed time of usage (day, night, weekend, etc.),
– maximum number of concurrently used resources.

Virtual Terminal Concept. With the increasing utilization of supercomputers for *interactive* use the support of remote access via WANs becomes more and more important. Unpredictable behavior and even temporary breakdowns of the network should (ideally) be hidden from the user.

In CCS, this is done by the EM which buffers the standard output streams (stdout, stderr). In case of a network break down, all open output streams are sent by e-mail to the user or they are written to a file. A user may re-bind to a lost session, provided that the application is still running. CCS guarantees that no data is lost in the meantime.

Worker Concept. In contrast to other parallel systems, each node in an SCI cluster runs a full operating system with all system functions and tools. Therefore, a wide range of software packages like debuggers, performance analyzers, numerical libraries, and runtime environments is available (Sec. 26.1.2).

Often these software packages require specific pre- and post-processing. For this purpose, CCS provides the so-called *worker concept*. Workers are tools to start jobs under specific run time environments. They hide specific procedures (e.g. starting of daemons or setting of environment variables) and provide a convenient way to start programs.

The behavior of a worker is defined in a configuration file (Fig. 26.5) by specifying five attributes:

```
pvm,                                                #name of the worker
%CCS/bin/start_pvmJob -d -r %reqID -m %island,      #run command
%CCS/bin/start_pvmJob -q -m %island,                #parse command
%root %CCS/bin/establishPVM %user,                  #pre-processing
%root %CCS/bin/cleanPVM %user                       #post-processing
```

Fig. 26.5. A worker definition for starting jobs in a PVM environment

- the name of the worker,
- the command for CCS to start the job,
- the optional parse command for detecting syntax errors,
- the optional pre-processing command (e.g. initializing a parallel file system),
- the optional post-processing command (e.g. closing a parallel file system).

Both pre- and post-processing can be started with either root or user privileges, controlled by a keyword. The configuration file is parsed by the user interface and can therefore be changed at run time. New workers can be plugged in without the need to change the CCS source code.

The *pvm-worker* used on our SCI cluster may serve as an example to illustrate what can be done with a worker: The pvm-worker creates a PVM host file (the host names were provided by the CM) and starts the master-pvmd. The master-pvmd starts, according to the given host file, all other slave-pvmds via the normal `rsh` or `ssh` mechanism to establish the virtual machine (VM) on the requested partition. Since the user application cannot be started until all pvmds are running, the pvm-worker starts a special PVM application after the master-pvmd is running. This little program then periodically checks how many nodes are connected to the VM until the entire VM is up. Thereafter the user application is started. After termination, the worker terminates the master-pvmd which shuts the VM down. The corresponding worker definition is shown in Fig 26.5.

26.2.3 Scheduling

In their resource requests, users must specify the expected finishing time of their jobs. Based on this information, CCS determines a fair and deterministic schedule. Both, batch and interactive requests are processed in the same scheduler queue. The request scheduling problem is modeled as an n-dimensional bin packing problem, where one dimension corresponds to the continuous time flow, and the other $n-1$ dimensions represent system characteristics, such as the number of processor elements. In general, CCS uses an enhanced first-come-first-serve (FCFS) scheduler, which fits best to the request profile in our center. Waiting times are minimized by first checking whether a new request fits into a gap of the current schedule (back-filling).

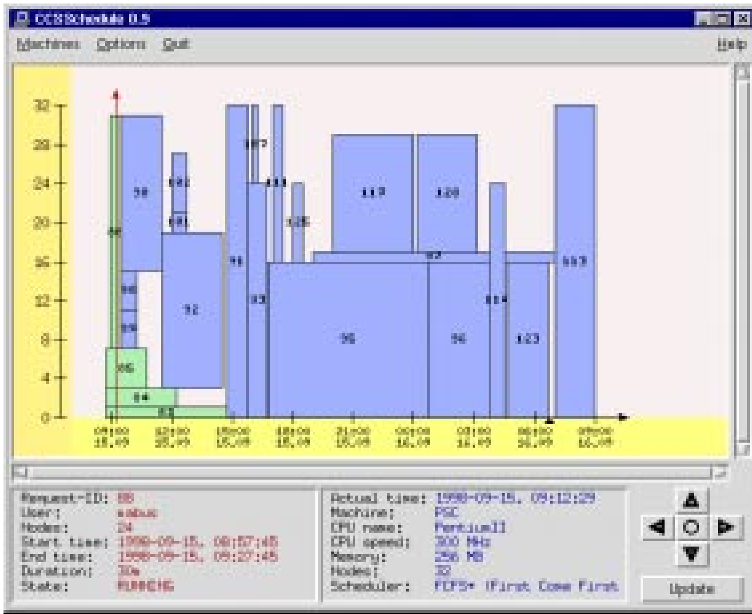


Fig. 26.6. Scheduler GUI displaying the scheduled nodes over time

Figure 26.6 depicts a typical view of the scheduler GUI. The schedule is displayed in an X-window that has been implemented with the portable Tcl/Tk package.

CCS provides several scheduling strategies (e.g., first-come-first-serve, shortest-job-first, longest-job-first), which can be chosen by the system administrator. The integration of new schedulers is easy, because the *Queue Manager* (QM) provides an API to plug in new modules. This also allows the QM to use several schedulers. At runtime, the QM makes a decision which scheduler to use, thereby being able to adjust to specific operating modes (e.g., interactive or batch).

With CCS, it is also possible to *reserve resources* for a given time in the future. This is a convenient feature when planning interactive sessions or online-events. As an example, consider a user who wants to run an application on 32 nodes of the SCI cluster from 9 to 11 am at 13.02.1999. The resource allocation is done with the command: `ccsalloc -m SCI -n 32 -s 9:13.02.99 -t 2h`.

Deadline scheduling is another useful feature. Here, CCS guarantees the job to be completed no later than the specified time. A typical scenario for this feature is an overnight run that must be finished when the user comes back into the office next morning. Deadline scheduling gives CCS the flexibility to improve the system utilization by scheduling batch jobs at the latest possible time so that the deadline can still be met.

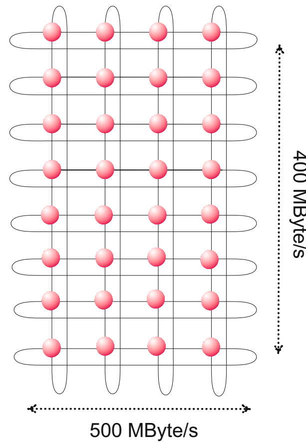


Fig. 26.7. Different link speeds of the 32-node SCI cluster at PC^2

The CCS scheduler is able to handle fixed and time-variable resources. A resource that has been reserved for a given time interval is fixed in time: it cannot be shifted on the time axis. Interactive requests, in contrast, can be scheduled earlier but not later than asked for. Such a shift on the time axis might occur when another user releases resources before the estimated finishing time.

26.2.4 Partitioning the System

On the one hand, we want to maximize the system utilization, whereas on the other hand, we wish to maintain a high degree of system independence for improved portability and easier adaptation to heterogeneous systems.

To deal with these two contradictory goals, we have split the scheduling process into two instances. The Queue Manager (QM) and the Machine Manager (MM). The QM is independent of the underlying hardware architecture [9]. It has no information on mapping constraints such as the minimum cluster size or the amount/location of entry nodes.

These machine dependent tasks are performed by the MM. It verifies whether a schedule given by the QM can be mapped onto the hardware at the specified time. If the schedule cannot be mapped onto the machine, the MM returns an alternative schedule to the QM.

This separation between the hardware-independent QM and the system-specific MM allows to encapsulate system-specific mapping heuristics in small code modules. With this approach, special requests for I/O-nodes, partition shapes, or memory constraints can be taken into consideration in the verifying process.

As an example, the *Mapping Verifier* (MV) in the MM takes system characteristics like the different speed of horizontal and vertical SCI links (see

Fig. 26.7) into account. With its more detailed information on the machine structure the MV employs system specific partitioning schemes. For our SCI cluster, it computes a partition according to the cost functions "minimum network interference by other applications" and "best network bandwidth for the given application".

The first function prefers to use as few SCI rings as possible, thereby minimizing the number of changes from the X- to the Y-ringlets in the 2D torus (Fig. 26.7), while the second function tries to map applications on single rings with maximum bandwidth.

The API of the MM allows to adapt the partioning to arbitrary topologies, or to implement mapping modules, that are optimally tailored to the specific hardware properties.

26.2.5 Job Creation and Control

At configuration time, the QM sends the user request to the MM. The MM then allocates the compute nodes, loads and starts the application code and releases the resources after the run. Because the MM also has to verify the schedule, which is a polynomial time problem, a single MM daemon might become a computational bottleneck. We have therefore split the MM into two parts (Fig. 26.8), one for the machine administration and one for the job execution. Each part contains several modules and/or daemons, which can run on different hosts to improve the performance.

The machine administration part consists of three separate daemons (MV, SM, CM) that execute asynchronously. A small *Dispatcher* coordinates the lower-level components.

The *Mapping Verifier (MV)* checks whether the schedule given by the QM can be realized at the specified time with the specified resources (see 26.2.4).

The *Configuration Manager (CM)* provides the interface to the hardware. It is responsible for booting, partitioning, and shutting down the operating system software. Depending on the system's capabilities, the CM may gather consecutive requests and re-organize or combine them for improving the throughput—analogously to a hard disk controller. Additionally, the CM provides external tools with information on the allocated partition, like host names or the partition size. With this information, external tools can create host files—to start a PVM application, for example.

One typical example for an external tool is the *CCS_MON* monitoring software. It collects CPU utilization, memory usage and other useful information on the status of the SCI nodes. Users may start a graphical frontend to watch the current status of their nodes. The CM provides the CCS_MON server with information on the node status via a UDP link. This information is then used by the CCS_MON user interface to highlight the corresponding frame(s) on the display (Chapter 25).

Ideally, a resource management system should provide all system features to the user, including permission to log into the owned nodes. But as a con-

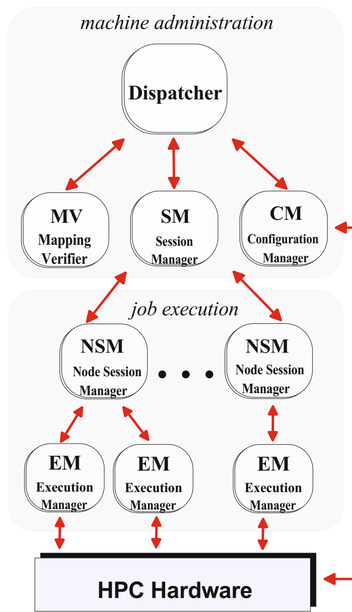


Fig. 26.8. Detailed view of the machine manager (MM)

sequence, users are then able to start arbitrary processes on arbitrary nodes and the system cleanup may become difficult. In CCS, this is the task of the *Node Session Manager (NSM)* which runs on each specified entry node with root privileges. The NSM starts and stops jobs and it controls the processes. At allocation time, the NSM starts an *Execution Manager (EM)* which establishes the user environment (UID, shell settings, environment variables, etc.) and starts the application. In space-sharing mode, the NSM changes the passwd file to avoid concurrent logins from other users. In time-sharing mode, the NSM invokes as many EMs as needed. It also gathers dynamic load data and sends it to the MM and QM where it is used for scheduling and mapping purposes.

The *Session Manager (SM)* synchronizes the NSMs. It sets up the session, including application-specific pre- or postprocessing, and it maintains information on the status of the applications. Figure 26.9 gives an overview of the control and data flow in a CCS island.

26.2.6 Reliability

With the transition from batch-oriented high-performance computing to interactive access, system reliability becomes an even more important issue because node breakdowns immediately influence the user's work flow. Additionally, today's parallel systems often comprise independent (workstation-

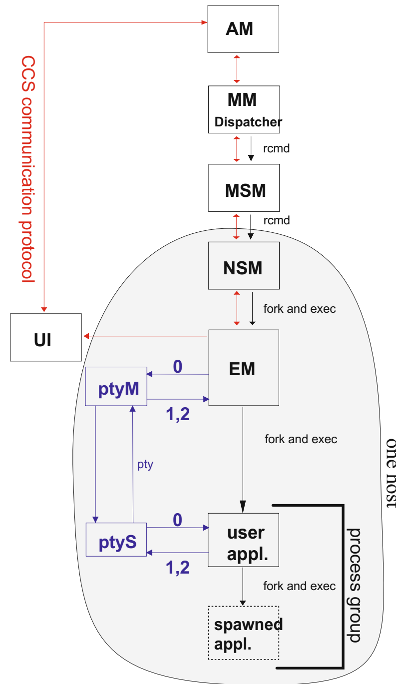


Fig. 26.9. Control and data flow in CCS

like) nodes, which are more vulnerable to breakdowns than the homogeneous nodes contained in a regularly structured parallel system.

A resource management system must be able to detect and possibly repair breakdowns at three different levels: the computing nodes, the software daemons, and the communication network.

Many failures become only apparent when the communication behavior changes over time, or when a communication partner does not answer at all. To detect a failure and to determine its reason (i.e. to decide whether the network is down or congested, whether the communication partner has died, or whether a node has crashed), the *Island Manager (IM)* maintains an information base on the status of all system components within the island. Each CCS daemon notifies the IM when starting up or closing down, so that the IM has a consistent view on the current system status.

When a CCS daemon detects a breakdown of another daemon, it closes the connection to this daemon and requests the IM to re-establish the link. The IM has a number of methods to find out about the problem. Which of them to use depends on the type of the target system. If it is an SCI cluster, the IM first tries to `ping` the daemon in question. If it does not answer in time, the IM then logs into the faulty node and tries to determine whether the daemon is still running or in which state it is. The IM is authorized to

stop erroneous daemons, to restart crashed ones, and to migrate daemons to other hosts in case of system overloads or crashes. For this purpose, the IM maintains an address translation table that matches symbolic names to physical network addresses (e.g. host ID and port number). Symbolic names are given by the triple <`site, island, process`> (e.g. PC2_PSC_MM). With this feature a cluster can be logically divided into several CCS islands, each of them with different scheduling or mapping characteristics.

If the IM cannot solve the problem, it sends an email with a problem report and the actions taken to the administrator.

For recovery purposes, each CCS daemon periodically saves its state to a disk. At boot time the daemons read their information and synchronize with their communication partners. This allows to shutdown or kill CCS daemons (or even the whole island) at any given time without the risk to loose requests.

26.3 Resource and Service Description

One important tool in CCS is the *Resource and Service Description RSD* [7]. It is used at the administrator level for describing type and topology of the available resources, and at the user level for specifying the required system configuration for a given application.

Compared to the text-based resource description language *RDL* [3] we have used in the early 90s, the new RSD scheme is more versatile. It has three interfaces:

– a GUI for specifying simple topologies and attributes,
– a language interface for specifying more complex and repetitive graphs (mainly intended for system administrators), and
– an API for access from within an application program.

The graphical editor stores the graphical and textual data in an internal data representation. This data is bundled with the API access methods and sent as an attributed object to the target systems, where it is matched against other hardware or software descriptions.

RSD data can only be accessed via the API. For later modifications it is retranslated into its original form of graphic primitives and textual components. This is possible, because the internal data representation also contains a description of the component's graphical layout. In the following, we describe the core components of RSD in more detail.

26.3.1 Graphical Representation

The graphical editor provides a set of simple modules that can be edited and linked together to build a hierarchical graph of the resource components.

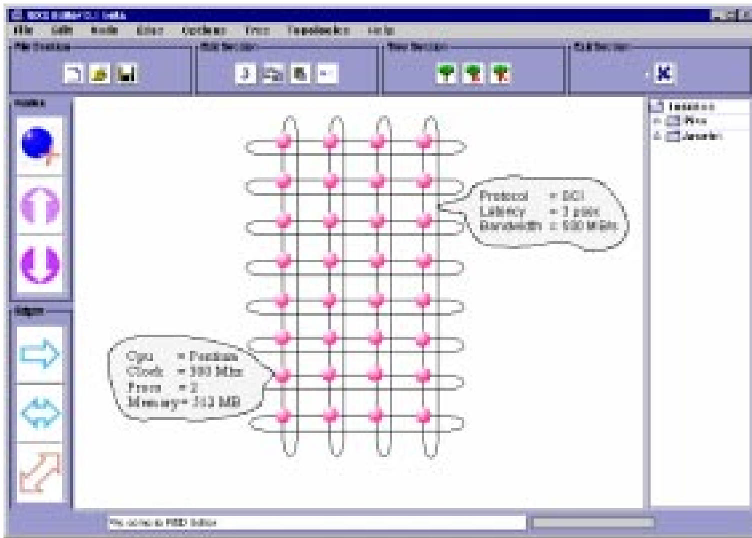


Fig. 26.10. RSD editor

System administrators use the GUI to describe the basic computing and networking components in a geographically distributed environment. Figure 26.10 illustrates a typical administrator session. The system components of the site are specified in a top-down manner with the interconnection topology as a starting point. With drag-and-drop, the administrator specifies the available machines, their links and the interconnection to the outside world. New resources can be specified by using predefined objects and attributes via pull down menus, radio buttons, and check boxes.

In the next step, the structure of the systems is successively refined. The GUI offers a set of standard machine layouts like Cray T3E or IBM SP2 and some generic topologies like ring, grid, or torus. The administrator defines the size and the general attributes of the system. When the system has been specified, a window with a graphical representation of the system opens, in which single nodes can be selected. Attributes like network interface cards, main memory, disk capacity, I/O throughput, CPU load, network traffic, disk space, or the automatic start of daemons, etc. can be assigned.

Users use the GUI for specifying their resource requests. There exist a number of predefined configuration files containing some commonly used resource descriptions. It is also possible to connect to a remote site for loading its RSD language dialect.

As an example, in one site the entity processor may be denoted by the attribute name "CPU", while another site may use the term "PE". Downloading a remote RSD dialect allows the user interface to perform online syntax checks when specifying remote resources. Likewise, it is possible to join mul-

tiple sites to a meta-site, using a different RSD dialect Note, that this does not affect the language used in the local sites.

26.3.2 Textual Representation

For some system administrator tasks, GUIs are not powerful enough for describing complex (meta-)computing environments with a large number of services and resources. Hence, we devised a language interface that is used to specify irregularly interconnected, attributed structures. Its hierarchical concept allows different dependency graphs to be grouped for building even more complex nodes, i.e., hypernodes.

Active nodes are indicated by the keyword `NODE`. Depending on whether RSD is used to describe hardware or software topologies, the keyword `NODE` is interpreted as a processor or a process.

Communication interfaces are declared with the keyword `PORT`. A `PORT` may be a socket, a passive hardware entity like a network interface card, a crossbar, or a process that behaves passively within the parallel program.

A `NODE` definition consists of three parts:

1. In the optional `DEFINITION` section, identifiers and attributes are introduced by *Identifier* `[ = (` *value*,... `) ] .`
2. The `DECLARATION` section declares all nodes with corresponding attributes. The notion of a 'node' is recursive. They are described by `NODE` *NodeName* {`PORT` *PortName*; `attribute 1, ...`}.
3. The `CONNECTION` section is again optional. It is used to define attributed edges between the ports of the nodes declared above: `EDGE` *NameOfEdge* {`NODE` *w* `PORT` *x* `<=>` `NODE` *y* `PORT` *z; attribute 1; ...*}.
   So called *virtual edges* are used to specify links between different levels of the hierarchy in the graph. This allows to establish a link from the described module to the outside world by exporting a physical port to the next higher level. These edges are defined by: `ASSIGN` *NameOfVirtualEdge* { `NODE` *w* `PORT` *x* `<=>` `PORT` *a*}. Note, that NODE *w* and PORT *a* are the only entities known to the outside world.

Fig. 26.11 shows the configuration of the 4 by 8 node SCI cluster operated in Paderborn. The corresponding resource specification is shown in Fig. 26.12. The cluster consists of two frontend systems and 32 compute nodes. The frontends are quad-processor systems with ATM connections for external communication and Fast Ethernet links for controlling the cluster and serving I/O of the applications running on the compute nodes. For each compute node, the following attributes are specified: CPU type, the amount of memory, and the ports of the SCI and Fast Ethernet network. All nodes are interconnected by uni-directional SCI ringlets in a 2D torus topology. The

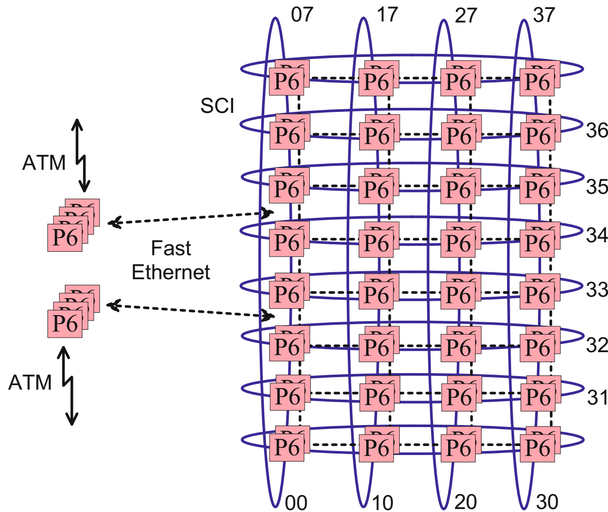


Fig. 26.11. Configuration of the 4 x 8 node SCI cluster in Paderborn

bandwidth of the horizontal rings is 400 Mbyte/s and 500 Mbyte/s in vertical direction. Each node is connected by Fast Ethernet to one of the frontends (gateways).

26.3.3 Internal Data Representation

The abstract data type establishes the link between the graphical and the textual representation of RSD. It is used for storing descriptions on disk and for exchanging them across networks. The internal data representation must be capable of describing the following properties:

– arbitrary graph structures,
– hierarchical systems or organizations,
– nodes and edges with valued attributes.

Furthermore, it should be possible to reconstruct the original representation, either graphical or text based. This facilitates the maintenance of large descriptions (e.g. a site with complex heterogeneous computing equipment) and allows visualization at remote sites.

In order to use RSD in a distributed environment, a common format for exchanging RSD data structures is needed. The traditional approach would be to use a data stream format. However, this would involve two additional transformation steps whenever RSD data is to be exchanged (internal representation into data stream and back). Since the RSD internal representation has been defined in an object oriented way, this overhead can be avoided, when the complete object is sent across the network.

```
 NODE PSC                  //This SCI cluster is named PSC
{
   // DEFINITIONS:
   CONST X = 4, Y = 8;      // dimensions of the system
   CONST N = 2;             // number of frontends
   CONST EXCLUSIVE = TRUE;  // resources for exclusive use only

   // DECLARATIONS:
   // we have 2 SMP nodes (frontends), each with 4 processors
   // each gateway provides an ATM port and a FastEthernet port
   FOR i=0 TO N-1 DO
      NODE frontend_$i {
      PORT ATM; PORT ETHERNET; CPU=PentiumII; MEMORY=512 MByte; MULTI_PROC=4;};
   OD
   // the others are dual processor nodes
   // each with a SCI port and a Fast Ethernet port
   FOR i=0 TO X-1 DO
      FOR j=0 TO Y-1 DO
         NODE $i$j {
         PORT SCI; PORT ETHERNET; CPU=PentiumII; MEMORY=256 MByte; MULTI_PROC=2;};
      OD
   OD
   // CONNECTIONS: build the SCI 2D torus, vertical direction
   FOR i=0 TO X-1 DO
      FOR j=0 TO Y-1 DO
         EDGE edge_$i$j_to_$i$((j+1) MOD Y) {
            NODE $i$j PORT SCI => NODE $i$((j+1) MOD Y) PORT SCI;
            BANDWIDTH = 500 MByte/s; };
      OD
   OD
   // CONNECTIONS: build the SCI 2D torus, horizontal direction
   FOR i=0 TO X-1 DO
      FOR j=0 TO Y-1 DO
         EDGE edge_$i$j_to_$((i+1) MOD X)$j {
            NODE $i$j PORT SCI => NODE $((i+1) MOD X)$j PORT SCI;
            BANDWIDTH = 400 MByte/s;};
      OD
   OD
   // CONNECTIONS: build the FastEthernet link to all nodes
   FOR i=0 TO X-1 DO
      FOR j=0 TO Y-1 DO
         EDGE edge_frontend_$(i MOD 2)_to_$i$j) {
            NODE frontend_$(i MOD 2) PORT ETHERNET <=> NODE $i$j PORT ETHERNET;
            BANDWIDTH = 100 Mbps; };
 };
```

Fig. 26.12. RSD specification of the SCI cluster in Fig. 26.11

Today there exists a variety of standards for transmitting objects over the Internet, e.g. *CORBA*, *Java*, or *Component Object Model COM+*. Since we do not want to commit on either of these, we only define the interfaces of the RSD object class but not its private implementation. This allows others to choose an implementation that fits best to their own data structures. Interoperability between different implementations can be improved by defining translating constructors, i.e. constructors that take an RSD object as an argument and create a copy of it using another internal representation.

26.4 Related Work

Much work has been done in the field of resource management in order to optimally utilize the costly high-performance computer systems. However, in contrast to the CCS approach, described here, most of today's resource management systems are either vendor-specific or devoted to the management of LAN- or WAN-connected workstation clusters.

The *Network Queuing System NQS* [14], developed by NASA Ames for the Cray2 and Cray Y-MP, might be regarded as the ancestor of many modern queuing systems like the *Portable Batch System PBS* [4] or the Cray *Network Queuing Environment NQE* [17].

Following another path in the line of ancestors, the *IBM Load Leveler* is a direct descendant of *Condor* [15], whereas *Codine* [10] has its roots in *Condor* and *DQS*. They have been developed to support 'high-throughput computing' on UNIX workstation clusters. In contrast to high-performance computing, the goal is here to run a large number of (mostly sequential) batch jobs on workstation clusters without affecting interactive use. The *Load Sharing Facility LSF* [16] is another popular software to utilize LAN-connected workstations for high-throughput computing. For more detailed information on cluster managing software, the reader is referred to [2, 12].

These systems have been extended for supporting the coordinated execution of parallel applications, mostly based on PVM. A multitude of schemes have been devised for high-throughput computing on a somewhat larger scale, including the Iowa State University's *Batrun* [19], the Dutch *Polder* initiative [8], the *Nimrod* project [1], and the object-oriented *Legion* [11] which proved useful in a nation-wide cluster. While these schemes emphasize mostly the application support on homogeneous systems, the *AppLeS* project [5] provides application-level scheduling agents on heterogeneous systems, taking into account their actual resource performance.

26.5 Summary

The *Computing Center Software (CCS)* is a resource management software for the user access and system administration of dedicated high-performance systems. It has been in operation since 1992 on various massively parallel systems and workstation clusters.

In principle, an SCI cluster can be managed just like an ordinary workstation cluster with, e.g. Ethernet network. But due to the much higher speed and throughput of the SCI links, we regard SCI clusters as dedicated, partitionable high-performance computers that are operated in multi-user mode. Especially the very large SCI clusters (with up to 192 PEs) operated at our site have more in common with traditional MPPs than with LAN connected workstations. Hence, we have adapted CCS for the management of SCI clusters. CCS provides:

– optimal space partitioning for concurrent access by multiple users,
– scheduling strategies known from high-performance computers,
– versatile tools (GUI, API) for specifying resources and services,
– reliability functions to support remote user access.

The modular concept of CCS proved very useful in our adaptation. We just needed to specify the topologies of our SCI clusters with the resource and service description tool RSD [7], and we had to implement new mapping modules for the optimal partitioning of the shared SCI links.

Acknowledgments

Thanks to the members of the CCS team, who have spent a tremendous effort on the development, implementation, and debugging since the project start in 1992: Bernard Bauer, Matthias Brune, Christoph Drube, Harald Dunkel, Jörn Gehring, Oliver Geisser, Christian Hellmann, Axel Keller, Achim Koberstein, Rainer Kottenhoff, Karim Kremers, Torsten Kuhnhenne, Fru Ndenge, Friedhelm Ramme, Thomas Römke, Helmut Salmen, Dirk Schirmer, Volker Schnecke, Jörg Varnholt, Leonard Voos, Anke Weber.

Also, special thanks to our collaborators at CNUCE, Pisa: Domenico Laforenza, Ranieri Baraglia, Mauro Michelotti, Simone Nannetti. The CCS project did not only benefit by the many fruitful discussions with the CNUCE team, but our Italian friends have also done a magnificent job in implementing the graphical user interface!

References

1. D. Abramson, R. Sosic, J. Giddy, and B. Hall: Nimrod: A Tool for Performing Parameterized Simulations using Distributed Workstations. *4th IEEE Symp. High Performance and Distributed Computing*, August 1995.
2. M. Baker, G. Fox, and H. Yau: Cluster Computing Review. Northeast Parallel Architectures Center, Syracuse University New York, November 1995. http://www.npar.syr.edu/techreports/index.html.
3. B. Bauer and F. Ramme: A General Purpose Resource Description Language. In: Grebe, Baumann (eds): *Parallele Datenverarbeitung mit dem Transputer*, Springer-Verlag Berlin, 1991, pp. 68–75.
4. A. Bayucan, R. Henderson, T. Proett, D. Tweten, and B. Kelly: Portable Batch System: External Reference Specification. Release 1.1.7, NASA Ames Research Center, June 1996.
5. F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao: Application-Level Scheduling on Distributed Heterogeneous Networks. *Supercomputing*, November 1996.
6. N. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.K. Su: Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro* vol. 15, no 1, Feb. 1995, pp. 29-36.

7. M. Brune, J. Gehring, A. Keller, and A. Reinefeld: RSD – Resource and Service Description. *Intl. Symp. on High Performance Computing Systems and Applications HPCS'98*, Edmonton Canada, Kluwer Academic Press, May 1998.
8. D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne: A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *FGCS*, Vol. 12, 1996, pp. 53–66.
9. J. Gehring and F. Ramme: Architecture-Independent Request-Scheduling with Tight Waiting-Time Estimations. *IPPS'96 Workshop on Scheduling Strategies for Parallel Processing*, Hawaii, Springer LNCS 1162, 1996, pp. 41–54.
10. GENIAS Software GmbH: Codine: Computing in Distributed Networked Environments. http://www.genias.de/products/codine, January 1999.
11. A. Grimshaw, J. Weissman, E. West, and E. Loyot: Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems. *J. Parallel Distributed Computing*, Vol. 21, 1994, pp. 257–270.
12. J. Jones and C. Brickell: Second Evaluation of Job Queueing/Scheduling Software: Phase 1 Report. Nasa Ames Research Center, NAS Tech. Rep. NAS-97-013, June 1997.
13. A. Keller and A. Reinefeld: CCS Resource Management in Networked HPC Systems. *7th Heterogeneous Computing Workshop HCW'98 at IPPS*, Orlando Florida, IEEE Comp. Society Press, 1998, pp. 44–56.
14. B.A. Kinsbury: The Network Queuing System. Cosmic Software, NASA Ames Research Center, 1986.
15. M.J. Litzkow and M. Livny: Condor – A Hunter of Idle Workstations. *Procs. 8th IEEE Int. Conference on Distributed Computing Systems*, June 1988, pp. 104–111.
16. LSF: Product Overview. http://www.platform.com/content/products/, January 1999.
17. NQE-Administration. Cray-Soft USA, SG-2150 2.0, May 1995.
18. F. Ramme, T. Römke, and K. Kremer: A Distributed Computing Center Software for the Efficient Use of Parallel Computer Systems. *HPCN Europe*, Springer LNCS 797, Vol. 2, 1994, pp. 129–136.
19. F. Tandiary, S.C. Kothari, A. Dixit, and E.W. Anderson: Batrun: Utilizing Idle Workstations for Large-Scale Computing. *IEEE Parallel and Distributed Techn.*, 1996, pp. 41–48.

Part IX

Perspectives

SCI has been—and surely continues to be—an intriguing success story because it has contributed to a new and promising field, the field of high-performance computing with commodity products. The final chapter shows the extent to which the development and industrial takeup of the SCI technology depends on individuals, technological trends, companies' interests, and sometimes even luck.

Who could be a better author for this kind of topic than David B. Gustavson, sometimes nicknamed "the father of SCI"? With much energy and enthusiasm, David was the chair of the IEEE Computer Society Microprocessor Standards Subcommittee on SCI, IEEE Std 1596-1992, and he is now executive director of the *SCIzzL Association* which brings together users, developers, and manufacturers of SCI technology.

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, p. 463, 1999
© Springer-Verlag Berlin Heidelberg 1999

27. Industrial Takeup of SCI and Future Developments

David B. Gustavson

SCIzzL/Santa Clara University
1946 Fallen Leaf Lane
Los Altos, CA 94024-7206
Email: dbg@SCIzzL.com

27.1 SCI's Cultural Context

The Scalable Coherent Interface began life in 1987, in the IEEE 896 Futurebus project. Futurebus was aiming at the high-performance bus/module market, above VME, hoping to be a multiprocessing backplane bus/module system.

Multiprocessing support in Futurebus evolved to include fair and prioritized arbitration; a set of primitives to handle process synchronization, locks and mutual exclusion; reflective memory, then cache coherence; and a variety of maintenance features as required for large systems.

Paul Sweazey, then of National Semiconductor, had led the Futurebus Cache Coherence task group, which developed the now-standard MOESI (Modified Owned Exclusive Shared Invalid) categorization of coherence schemes. After having finished that task, he took time to think about where things were going.

Sweazey extrapolated microprocessor chip performance versus time, and showed that within just a few years Futurebus would be unable to support multiprocessing meaningfully, because one processor would be able to saturate the bandwidth of the fastest backplane bus possible. Moreover, ever-increasing levels of integration had moved computer systems from rooms to racks to crates to boards, and soon to chips.

Thus the whole concept of a high-end modular bus had a finite remaining lifetime. Buses would be useful for I/O adapters, for customizing a system, but would be inadequate for memory expansion or multiprocessing demands.

Sweazey then chaired a Study Group under the IEEE Computer Society's Microprocessor Standards Committee, which was then chaired by Bob Davis, to consider what to do about this problem. After just a few meetings, the outlines of a potential solution began to appear, and a formal IEEE Project was begun, chaired by me.

At this point, all our emphasis was on the very high end of possible performance, and cost was a secondary consideration. Furthermore, we had to define our territory to avoid damaging the market for Futurebus. The agreement we worked out with Futurebus chairman Paul Borrill was that SCI's goal would be 1 GByte/s per processor, far above anything that even

H. Hellwagner, A. Reinefeld (Eds.): SCI'99, LNCS 1734, pp. 465-480, 1999
© Springer-Verlag Berlin Heidelberg 1999

the most optimistic Futurebus enthusiast thought any backplane bus could ever do. (Some buses have now exceeded 1 GByte/s in total, but only by reducing loads and distances to far less than a backplane module/bus system requires.)

At about that time, Futurebus 1987 was complete but lacking much commercial support. The US Navy decided that Futurebus was just what it needed, if only a few minor changes would be incorporated. Therefore, the Futurebus+ group was formed and began to revise the 1987 specification, to add more priority levels and more performance and packaging options. The effect of this effort was to kill adoption of Futurebus 1987 because its replacement was imminent. However, working out these modifications took several years, so that Futurebus+ did not get approved until 1991.

IEEE 1394 SerialBus (now sometimes commercially called FireWire^{TM} or iLink^{TM}) was being developed at the same time, by some of the same people, and was incorporated into Futurebus+ and SCI and several other bus standards as an alternate path for diagnostics and I/O. There was a deliberate decision to optimize SerialBus strictly for low cost for desktop I/O, without regard for scalability or the cost, complexity, or performance of future switches/bridges. SerialBus did adopt the same address space model as SCI, and a similar command set, though.

SerialBus, Futurebus, and SCI also share a co-developed Control and Status Register architecture, IEEE Std 1212, chaired by SCI architect David James. This was a painful laborious process, because though his interests lay mainly toward the scalable SCI, it was important to keep the other groups committed to this joint architecture. That meant in effect that every complaint from the other groups was equivalent to a veto of the project, so every issue had to be worked through carefully, both with regard to their desires and also with regard to problems those desires might introduce when scaled up. Eventually agreement or respectful compromises were reached on all points and the standard was completed.

The marketing model in the early days of SCI development showed VME gaining market volume until Futurebus+ products arrived, then gradually declining as Futurebus+ volumes grew, and after about 10 years SCI would appear and begin to take volume from Futurebus+. That was attractive for everyone, because it followed the expected natural progression, allowed for planning, and reassured the customer that there was no dead end in sight.

Unfortunately, that is not how things turned out. The delay in producing Futurebus+ resulted in a large increase in microprocessor power, which nearly closed its multiprocessor-application window, and VME continued to develop extensions that increased its performance so VME could handle the remaining application space adequately. Thus there was no compelling application to justify the costs of adopting a completely new technology like Futurebus+. A few companies shipped Futurebus+ products, including DEC, but the economics did not work anymore. (For example, to keep bus stubs

as short as possible, Futurebus+ had to use a very wide bus with transceivers as close to the connector as possible, resulting in many packages and many pins—17 packages in one commercial version. The protocol variants also made the controller chip rather complex, the asynchronous protocol was hard to test under worst-case conditions, and the need for short stubs made it impossible to use a passive extender card to make boards accessible while trouble-shooting. Thus trouble-shooting tended to change the signal timings, which often changed the conditions that were to be studied.)

On the other front, SCI arrived far too early. Futurebus+ was difficult partly because there was complete freedom how to do everything, so it was hard to agree which of several methods should be used. (In some cases there was no agreement, and more than one method was adopted by the standard, deferring the decisions needed for product compatibility to other documents, which were called “profiles.”) Futurebus meetings were very large, with many participants from every part of industry, so technical work was difficult, procedures became more formal, and decisions were by vote. Backtracking in the design space after problems were discovered was too difficult.

SCI also had complete freedom, but its technical problems were very difficult and there was seldom more than one good solution to choose from. Also, SCI had no credibility, due to its preposterous goals, so most people attended only because of personal interest rather than corporate agenda, and the meetings were small. As a result, it was possible to work in a different way that would avoid the dynamics that plagued Futurebus (and its predecessor IEEE 960 Fastbus). We had enough people, often the leading experts in the field, but not too many. We had leading researchers from academia, and chief architects from computer manufacturers. We asked for advice from companies active in the multiprocessor field, such as BB&N and Sequent, and they were extremely helpful. Norsk Data, later Dolphin Server Technology, planned from the beginning to build computers using SCI as the processor/memory interconnect, so they worked intensively with us and had a major influence in keeping the SCI design sound and practical.

Furthermore, we were fortunate to have the right personality mix, so that we could operate in a no-holds-barred ego-free nonpolitical environment. And, we had several key participants available fulltime, especially our technical editor and chief architect David James, of HP and later Apple Computer.

The Internet also helped us work faster. HP provided an FTP server that could be accessed by all participants, and a reliable platform-independent document formatting program, FrameMaker, became available. This made it possible for people in the US to perform a round of edits, then pass the document over to people in Europe for further work, and vice versa on a daily basis when appropriate. The nine hour time difference made it possible to essentially double the editing hours available without having to deal with multiple-copies/multiple-authors problems. Later the HP server was replaced

by one at Santa Clara University and then by one at SCIzzL, and FrameMaker was supplemented by another versatile technology, Adobe Acrobat.

There was only one vote in the SCI working group. That was the vote at the very end, to declare the project complete. Up to that point, every decision was made by consensus. If the right answer was clear, we took it. If not, we took a "good enough" answer until we ran into trouble, then we would backtrack if needed.

As a result, technical progress was extremely fast, with a document that was essentially complete in 1990 but was polished and refined until 1992. This schedule was unexpected—none of us would have predicted that SCI would take far less time than Futurebus. We started the project fully expecting to spend 10 years or more.

This schedule anti-slip threatened the marketing model of Futurebus, which caused a great deal of distress and even personal hostility.

But the threat was even worse than it first appeared, because SCI had to be very simple in order to run at such high speeds, because SCI had no need for separate transceiver chips or terminators or constrained physical layouts (e.g. short bus stubs), because SCI fit easily within one corner of an ASIC and needed only 72 pins for 2 GByte/s, and because SCI's bandwidth scaled up with system size, and worked with fiber optic links as well as copper cables.

So the cooperative "VME then Futurebus then SCI" marketing model vaporized, and SCI became opposed by a large number of influential people whose goals were threatened.

Another marketing problem was the lack of ego commitment to SCI. Usually, for an interconnect standard there will be attendees from a large number of companies. The attendees will see numerous problems that are not being addressed to their satisfaction, and will join in the work, make some contributions, and thereby become committed to its success. They lobby their company management for support and commitment, and build up a substantial visibility in the marketplace. This can be essential for getting the critical mass of support needed for the success of a standard. (Of course, it is also responsible for many of the baroque features found in most standards, and adds a great deal of complexity to a system.)

We had a steady stream of visitors attending the monthly multi-day meetings, but few developed that sort of attachment. Our work was often very esoteric, such as working through the requirements of forward-progress guarantees (no deadlocks or livelocks/stalls) and their implications for the cache coherence mechanism. Furthermore, we had bypassed many of the more tractable problems by adopting other current work. For example, we adopted as a whole the new metric crate and backplane system (IEEE Std 1301) that had been developed primarily for Futurebus (with only one P1301 participant representing SCI interests).

As a result, we had only a few committed companies when we finished the work, not a broad base of industry acceptance. Our main competition at that

time seemed to be Futurebus, with many committed supporting companies and significant visibility in the market and the trade press; and FibreChannel, with perhaps even greater commitment and certainly very professional coordinated marketing.

SCI's image as a backplane bus/module system also broke down for another reason. If manufacturers have a choice between using their favorite packaging for their module designs (connecting the modules by cable), and redesigning their modules to fit some standard's requirements, most prefer to use their own packaging. It was the severe constraints of backplane buses that caused standard modules to be widely used. Once SCI removed those constraints, allowing arbitrary packaging connected by cables, virtually all designs chose something other than SCI's recommended mechanical crate/module standard. Thus SCI devices have no common appearance that can be used for marketing photos, and SCI received very little promotion from the module and crate manufacturers, a significant reduction of visibility in the marketplace compared to a normal bus standard.

But probably the biggest obstacle to marketing SCI was the lack of a simple way to explain what it is, what it can do, how it relates to well-known and understood products. SCI began as a high-end backplane bus for processors and memory and I/O, but became much more general, supporting cables and switches and acting somewhat like an incredibly fast local area network, and simpler/cheaper. It required much more explanation than a typical new product that merely lowers prices or runs a little faster.

Too late, some better names were invented, like Local Area Multiprocessor and Local Area Memory Port, but by then "SCI" had gained its own name recognition. More recently, the descriptive term "concurrent bus" has begun to look interesting—more on that later.

27.2 SCI Marketing and Adoption

SCI and FibreChannel began marketing at almost the same time, but FibreChannel was enormously well funded and well coordinated, supported by large companies with significant resources, which saw FibreChannel as useful for solving real problems and also as a potential source of sales.

SCI lacked that kind of support. Only one company, Dolphin (no longer in the processor business), promoted SCI as an open interchange standard. The Navy did adopt SCI as the unifying interconnect for its Joint Advanced Strike Technology (JAST) future fighter program, and had an interest in SCI's commercial success because of the need for military designs to take advantage of cost savings by using Commercial Off-The-Shelf (COTS) products wherever possible, and this did add somewhat to SCI's visibility. Unfortunately, it also created enemies, because JAST was seen as an important design win for FibreChannel as well. There were a number of design competitions and "shootouts" as a result, and last I saw FibreChannel had gained approximately

equal authorization. The program has now gone under wraps, so the current status of SCI in JAST is not obvious, but indirect evidence indicates that SCI is still included. (For example, one startup company, iCore Technology, has received some SBIR (Small Business Incentive) funding from military sources, to bring a high performance PCI/SCI interface to a manufacturing-ready state.)

FibreChannel shared one marketing problem with SCI, that of being hard to describe or categorize, because it combined networking features with channel-based I/O systems, which placed conflicting demands on switches (channels prefer circuit switching and networks prefer packet switching). Initially, FibreChannel systems all required switches; however, after a presentation about SCI (perhaps the timing was a coincidence) a new project was initiated to define a ring for FibreChannel too, called the FibreChannel Arbitrated Loop.

The other companies that adopted SCI mainly did so in order to take advantage of its technology for high performance rather than for low cost, and most had no interest in using it as an open interchange standard enabling them to interconnect with others' products.

Another obstacle is inherent in the nature of SCI: SCI was designed to be integrated directly into ICs that are mainly doing something else, such as processors, I/O controllers, switches, and memory controllers. Previous buses created a market for transceiver chips, which met the bus standard on one side and used conventional logic signals on the other. But conventional logic's external signals are not fast enough to keep up with SCI, so the back end of an SCI transceiver has to use a wider data path than the SCI links do.

The Working Group should have gone on to specify a standard for such a back-end bus, but did not. There was a reluctance to standardize yet another bus as part of SCI, since the design process for the SCI standard was driven by the problems inherent in such buses. Following that logic always ends up moving SCI right into the customer's ASIC. But that makes it hard to get started—we were too early for solving the problem by putting SCI into every vendor's ASIC cell library.

Three companies designed (for sale to the public) different chips that interfaced SCI to other circuits via back-end buses: Dolphin Interconnect Solutions [5], Vitesse Semiconductor [18], and Interconnect Systems Solution [7]. The production versions of these did converge on the official standard, though not all at the full SCI speed. The first product from Interconnect Systems Solution uses an 8-bit-wide link (IEEE Std 1596.3) and runs at 100 MByte/s in order to take advantage of low cost CMOS processes. This product was actually designed for a particular customer, much of whose application (including a processor!) is integrated on the same chip (in the true spirit of SCI), but it can also be packaged for general sale with the custom portion disabled.

In 1994, I left the Stanford Linear Accelerator Center, which had supported my work on SCI and related standards, and formed a new organization called SCIzzL at Santa Clara University, at the invitation of Prof. Qiang Li. SCIzzL (pronounced “sizzle,” or like “scissors,” the other cutting-edge tool that’s safe to use) is a partial acronym abbreviating something like “Scalable Coherent Interface Local Area Multiprocessor Users, Developers, and Manufacturers Association,” which did not seem to lend itself to any catchy and concise acronym, notwithstanding various attempts at substitution and reordering.

SCIzzL was envisioned as providing a focus for the formation of an SCI trade association, and as a mechanism for supporting ongoing standardization work in the IEEE to enhance SCI and its relatives and descendants. Funding for SCIzzL relies wholly on memberships or donations from interested companies, which turned out to be a problem due to the nature of the SCI market until now. Fortunately, one of those supporting standards, IEEE Std 1596.4 RamLink, became the initial focus of the memory industry’s effort to create SLDRAM (Synchronous Link Dynamic Random Access Memory, P1596.7) as an alternative to the Rambus RDRAM memory devices, which it was feared would put Intel in control of the memory industry. For several years this effort provided ample support for SCIzzL, but also a significant distraction from SCI. In 1999 the SLDRAM was abandoned, though it had demonstrated technical success with two independent working device designs, because Intel had made clear its commitment to RDRAM. SLDRAM Inc. was renamed “Advanced Memory International, Inc.” and became a marketing and coordinating organization for the remaining DRAM industry, and SCIzzL divorced from that group, to find out whether the SCI marketplace has yet grown enough to begin supporting further standards development, and a users association, on its own.

The advanced signaling technology developed for the SLDRAM bus has become SLIO in JEDEC, and will apparently be used for the second generation of Double-Data Rate DRAMs (DDR2). SLIO has individually adjustable high and low levels for drivers, fine adjustment of individual bit timing, and the option of using stub-decoupling resistors (compensating for the attenuated driver signal amplitude with the individual adjustments as necessary). The signaling is mostly single-ended, but with differential clocks.

Another part of SCI that has become widely adopted is IEEE 1596.3 LVDS, Low Voltage Differential Signals. This standard was deliberately scoped to apply specifically to SCI (otherwise the whole world would have piled on and it might never have finished due to territorial disputes etc.). Almost immediately it was generalized and specified for standard telecom industry rates, like 622 MBit/s, as TIA/EIA-644. A chip-enable has been added by some vendors, with double the current drive, in order to allow using LVDS in a bus configuration rather than point-to-point. LVDS has become the de facto standard for connecting flat panel displays in laptop computers.

GLVDS, a more advanced version developed and used by Ericsson Telecom is now being standardized in EIA/JEDEC Committee 16. This has signal swings similar to LVDS (0.25 to 0.5 V) but has the low level near ground (the G of GLVDS) instead of having the center value above 1 V as LVDS does. At the time LVDS was defined, its levels seemed adequate for several generations of power supply voltages and several chip technologies, and CMOS manufacturers thought they needed some voltage above ground to make the chips practical. GLVDS also defines the common-mode termination scheme, which was left unspecified in 1596.3.

27.3 Commercial Adoption of SCI

27.3.1 Interface Chips and Products

Dolphin Interconnect Solutions. Dolphin began as a minicomputer company, Norsk Data, then reorganized as Dolphin Server Technology when Norsk Data withdrew from that business, and finally abandoned the server market entirely, for an interconnect business based on SCI. Dolphin has experimented with several strategies along the way, and has had to discover the difficult tradeoffs between being a chip supplier and being a board or subsystem vendor and risking competition with its own customers. Some potentially large customers, e.g. Bit 3 Computer, were frightened away from SCI because of the prospect of competing with their sole supplier of SCI chips, which I believe was one of the biggest setbacks for SCI market acceptance in the early years. Dolphin is sensitive to this problem, and is increasingly supportive of growing the market as a whole rather than maintaining a near monopoly.

Dolphin's products include CMOS SCI interface chips, interface boards, switches, and development tools. See their web site [5] for current information; in particular, check the application notes at [6].

Dolphin Interconnect Solutions is still by far the dominant producer of SCI interfaces.

Vitesse Semiconductor. Vitesse Semiconductor builds a GaAs SCI interface for Sequent Computer (now being owned by IBM), but has not succeeded in marketing it broadly. This was the first practical chip to run at full standard speed (1 GByte/s). I think most customers are reluctant to use GaAs, regarding it as exotic, and they do not think they really need GByte/s bandwidth. I think a good strategy would have been to show that this system can economically provide excellent performance for a large number of attached devices that have ordinary bandwidth requirements, without needing an expensive switch. In other words, ignore the link bandwidth and focus on the cost for n-port interconnect performance based on a simple ring, with the built-in safety feature that switches can be added later as requirements

increase. Vitesse says the chips are still for sale to other customers, but a manual search of their Web site today reveals no mention of SCI or of those chips, and recent private comments from the management indicate that they have given up expanding this market.

Interconnect Systems Solution. ISS was founded by Khan Kibria, the lead designer of an SCI chip being designed at Unisys with other companies, including Vitesse. ISS does custom SCI (and other) chip design, but with an eye to leveraging the custom work to make SCI parts available to others. The ISS business model has been to support growth out of sales, which has hindered rapid growth. What is needed for making these chips broadly available is an investor who would fund a production run, and then sell the chips to various customers. The first chips use an 8-bit-wide SCI link and run at 100 MByte/s per link. When packaged for sale as general purpose chips, they use a simple wide back-side bus and internal features are memory-mapped for easy access in a wide variety of applications.

Lockheed Martin. Lockheed has developed a fast 16-port switch called RelianetSCI; see [9].

27.3.2 Coherent Shared Memory Implementations

Convex/Hewlett Packard. The first computers to use SCI as a high-performance cache-coherent shared-memory (CC-NUMA) multiprocessor interconnect were the Convex Exemplar supercomputers. Multiple SCI rings were used in parallel to increase bandwidth, connecting switch-based hypernodes of 8 PA-RISC processors. The SCI chips started from a Dolphin design, but were optimized by Convex and fabricated by Fujitsu in GaAs.

The Exemplar allowed writing programs on an HP workstation, and transparently scaling them up for parallel execution by re-compilation, gradually optimizing the program for more efficient operation across a larger number of processors. This is the beauty of coherent shared memory compared to message-passing for multiprocessing. On the other hand, conversion from a single workstation environment to a message-passing cluster environment usually requires a comprehensive rewriting and reorganization of the software.

Hewlett Packard bought Convex and added these machines to the top end of its line of workstations. Further optimization took place in subsequent generations of the product, and Convex/HP has not made the SCI signals available for external connections, so there has been no motivation for them to adhere to the standard—SCI was just a cheap source of useful technology.

Convex did make several attempts, in partnership with the DOE's Stanford Linear Accelerator Center, to get funding from the Department of Energy or the Department of Defense, to build an SCI-standard interface for external use. This would have been tested in the demanding High Energy Physics particle-accelerator laboratory as a tool for rapid data acquisition and analysis. Though the specifications of the proposed Convex SCI system

seemed like a perfect fit for several DOE and DOD problems, no funding was ever forthcoming. This was probably partly an accident of bad timing, as the DOD was then in the process of commercializing its own funded technology, Myrinet. If Myrinet succeeded commercially, the government funders could easily justify their continued existence as a government program. But if it was surpassed by some other technology, they would not look so good.

Sequent Computer. Sequent specializes in high-end multiprocessor servers for commercial applications, mainly transaction processing. They had used a highly optimized backplane bus along with very carefully tuned software in order to get high performance for particular classes of application, but they had reached the fundamental limits of that technology.

The Sequent NUMA-Q products are SCI based. Sequent no longer builds any bus-based systems.

The first generation of NUMA-Q did not take full advantage of SCI's potential speed, opting instead to minimize risk by allowing easy design updates via firmware uploads. Subsequent generations were able to proceed with confidence into faster hardware implementations. Sequent has not made the SCI signals available for external use, so once again there is little motivation to adhere to the standard in all respects, and a variety of specific optimizations have undoubtedly been incorporated. However, the SCI links have been implemented with Vitesse GaAs chips, which have also been available for purchase by others, so the low level protocols are presumably quite close to the standard.

The Sequent systems are based on an SMP Quad of Intel processors, four processors sharing a common memory via a local SMP bus. Sequent converts between the Intel MESI and the SCI coherence protocols in their interface to this bus. The Quad memory occupies a fraction of the global address space. Sequent's operating system, "ptx," is spread across as many as 16 Quads. More details can be found in White Papers at [13]. (In particular, see: "NUMA-Q: An SCI based Enterprise Server", "Implementation and Performance of a CC-NUMA System", and "Considerations in Implementing a System Based on SCI".)

An overview of the NUMA-Q 2000 can be found at [12]. The NUMA-Q 1000 (see [11]) uses a simplified version of the SCI protocol for a two node system, but does not use the SCI physical layer.

In 1999, IBM announced that it would buy Sequent to fill in the high end of its server line.

Data General. Data General uses SCI in its high-end AV20000 and AV25000 servers, referring to it as their "NUMALiiNE Technology." Their approach is similar to Sequent's, using quad Intel processors as the building block, and designing their own coherent interface from the quad's bus to SCI. Data General uses the Dolphin SCI interface chips, so is very close to the standard, but has not made the SCI links accessible outside the machines as yet. However, rumor has it that DG envisions using SCI eventually as an open

interface for third party I/O devices, which might become the first true use of SCI as an interchange standard as opposed to just being a cheap source of high performance technology for internal use.

More details about the Data General machines can be found in their White Papers at [3]. Other articles of interest include [2] and [4].

27.3.3 Non-coherent Implementations

Cray Research. Cray Research studied SCI and adopted a variant of it in 1995 as the GigaRing I/O system for the three larger families of Cray mainframes, the J90, T90 and T3E MPP plus a suite of I/O subsystem modules including FibreChannel disk arrays, HIPPI channel adapters, ESCON and Block-Mux tape channel adapters and a Multiple Purpose Node (MPN) adapter based on the SPARC processor and SBus technology for various other peripheral and network adapters. The GigaRing I/O system provided a common I/O product base for the mainframe products as well as a flexible system interconnect between mainframes or shared I/O.

Because of production considerations, to fit into qualified chips, it was necessary to double the width and halve the speed of standard SCI. In addition, encoding was added so that ground-potential differences could be blocked with high-pass capacitive coupling, and dual counter-rotating rings were used in order to support live removal and insertion of devices or systems, as well as for higher performance. Since compatibility with the standard was not a consideration, Cray also added protocol features to support multiple virtual channels.

IBM. IBM's AS400 designers began looking for a way to "firmly couple" systems in about 1991, discovered SCI and decided that SCI's direction looked about right. Though the original motivating project didn't develop as expected, IBM built a test chip in BiCMOS and described it in a 1995 ISSCC paper. This paper concluded that the SCI technology was robust and manufacturable. Eventually an 8-bit-wide version was designed for use as a mezzanine bus for I/O, the RIO interface, which is now shipping in AS400 and RS6000 machines. The physical signaling is similar to SCI's LVDS, and the interfaces implement per-bit deskew. IBM was the first to implement and validate SCI's per-bit deskew scheme. Like SCI, the signals in these parallel links are not encoded for DC balance. IBM favors longer CRC codes than SCI's 16 bits; 32 bit CRC was used for the test chip.

Some of the same IBM designers are now working on the physical layer for FutureIO; the FutureIO links look rather similar, but probably will include encoding for DC balance to simplify ground-isolation in large systems.

Scali. Scali began as a result of experience with military signal processors at Kongsberg. The plan was to demonstrate equivalent or better performance to custom-built signal processors by using COTS technology, and the strategy was to do this by connecting processors with SCI.

Scali began with building blocks of dual SPARC processors sharing dual SCI interfaces (via Sbus). With two 4-port switches from Dolphin, this allows 8 processors to be interconnected redundantly for reliable operation.

The next family of products was based on Intel processors connecting to SCI interfaces via the PCI bus. In 1998 a 64- and a 192-processor system were delivered to Paderborn [8]. With two SCI interfaces on a PCI card, Scali implemented a 2-dimensional toroidal interconnect, which does not require any switches [10]. The Scali technology is now the basis for the Siemens hpcLine computers [16].

Siemens. Siemens uses SCI to connect large numbers of PCI and other buses in the enhanced I/O system for its RM600E processors [17]. In 1999, Siemens began to market Scali machines as building-block components for building large systems, called the hpcLine [16].

Sun. There are two main applications of SCI at Sun as of 1999, clustering and high-performance computing, with other applications expected soon.

Clustering is currently limited to 2, 3 or 4 nodes (using the Dolphin 4-port switch). The hardware is made by Dolphin, the software mostly by Sun.

The low end is the Ultra 2 desktop, with Sbus interface. At present, only Sbus machines are shipping, but PCI is coming. Currently, most of Sun's Enterprise Servers (Starfire) have Sbus, but PCI models will be available soon.

There will also be a PCI-based Workgroup Server E250, and a bigger one, E450, that support SCI, so essentially the whole server product line will soon be supporting SCI.

Auspex. Auspex makes high-end network storage servers, called 4Front, based on SCI; see [1].

Silicon Graphics. Silicon Graphics Inc. has been a strong exponent of coherent shared memory, and has made a very strong case for SCI on several occasions (but without mentioning SCI; see [14]). However, SGI chose to define its own interconnect, most recently called a System Area Network, formerly known as Super-HIPPI. As of 1999, I have the impression that SGI was not dominant enough in the marketplace to succeed with this strategy.

SGI bought Cray Research just as Cray was implementing its SCI-like GigaRing, but did not stop Cray from proceeding with the GigaRing deployment. At the time of this writing, SGI is cutting back severely, and is reported to be ready to sell Cray again.

27.4 Future Directions

There are several high speed signaling technologies (e.g., Gigabit Ethernet and FibreChannel) and at least two I/O architectures (NGIO and FutureIO) that are beginning to move into SCI's territory.

NGIO and FutureIO appear at first blush to be doing essentially identical tasks—they interface to a processor at its full-bandwidth nexus, the "North Bridge," and connect to a wide range of I/O devices with high bandwidth. But they do not solve all the problems—they do not have shared memory!

Shared memory is highly desired by the users of multiprocessor systems who want high performance, because it can provide interprocess communication latencies that are about 100 times better than channel-based communication for a given technology. (With shared memory, communication is a part of one memory reference instruction, instead of the many instructions needed by non-shared-memory methods to set up buffer content, start the data moving, and extract it at the receiving end.)

The hard part of providing shared memory connections (e.g. for use by SCI) has been getting full access to the processor bus, which both NGIO and FutureIO will solve.

The real problem is one of business strategy. If Intel allowed NGIO to use its access to the processor bus to support coherent shared memory, then anyone could build large powerful multiprocessors by stacking arrays of cheap high-volume processors, which would wipe out the high-profit high-end market for special expensive processor chips that has been carefully guarded until now.

Only a few companies have been allowed to build their own interfaces to Intel processors at the full processor-bus level, i.e. Sequent and Data General, for use in their shared memory high-end servers using SCI. Another approach, by Corollary, was limited to smaller systems, but anyway Corollary has now been bought by Intel.

27.4.1 IEEE P2100 (SerialPlus)

The original SCI architects have been looking at designs for the follow-on generation for SCI ever since SCI's completion. This process has gone through several generations of complete ground-up redesign, while waiting for the right time, the right customer, the right marketing strategy, to make bringing out the next generation a useful exercise. There is a balance to maintain between obsoleting existing products, which can kill a standard, and letting the technology stagnate, which can also be fatal. The place where this work has been most visible has been IEEE P1394.2 Serial Express, later renumbered as P2100, and now with a tentative new name, SerialPlus.

Although these protocols are capable of doing everything SCI does, and more, the SerialPlus document is positioned for serial links of modest speed, and with the capability of encapsulating whole SerialBus packets. This is to take advantage of a possible market opportunity for providing a truly scalable backbone (and more) to allow extending SerialBus systems arbitrarily. So, cache coherence and parallel links are barely mentioned, but behind the scenes the protocols have been designed to support those extensions smoothly, and

to add some useful features to SCI. The following paragraphs reflect this new positioning in the presentation of SerialPlus.

SerialPlus is a concurrent bus, which can do many things at once. Like NGIO and FutureIO, SerialPlus devices can be connected by cables to switch hubs. However, in addition SerialPlus allows devices to be connected as a daisy-chained cable bus, so that many devices can share a switch port, or not use a switch at all. This makes SerialPlus much more versatile, gives it a much lower entry cost, and allows much better balancing of device requirements against switch port costs.

SerialPlus represents several generations of refinement of the proven SCI technology, to make it more consumer friendly, more robust, more versatile, and more economical. At present SerialPlus is being positioned by its designers as a useful backbone for interconnecting IEEE Std 1394 SerialBus devices (digital video cameras etc.), to get past the 1394 length and bandwidth limitations. (There are also other IEEE projects working on these problems, P1394.1 and P1394b.)

The advantage of SerialPlus is its scalability. Even though it will probably start at 1 GBit/s speeds for 1394-interconnecting applications, its protocols scale up to any speed that technology can offer, and a SerialPlus system can scale to any size or performance requirement by adding additional cables and switches. SerialPlus eases backward compatibility for future devices by supporting multiple speeds, intermixed. It supports isochronous transfers (guaranteed delivery time as required by audio/video data), live insertion/removal of devices without disturbing other users, discovery protocols, automatic hardware error recovery, nonstop robustness or failover by means of redundant paths and redundant packet streams, and arbitrary connection topology.

27.4.2 Concurrent Buses—A New Name for this Technology

SerialPlus (like SCI) is a *concurrent* bus. Other buses only allow one device to transmit at a time, but a concurrent bus allows any number of devices to transmit at the same time. The signals in the bus cables are directed so that these transmissions do not interfere with each other, and no information is ever lost. Each device cooperates by storing any information it receives while transmitting, and then sending that information along as soon as possible. I.e., only one signal at a time can be present on a particular piece of cable, but different signals can be active on every piece of cable at the same time.

Obviously this means that, under the covers, SerialPlus cannot really be a bus. Real buses have continuous unbroken signal wires from one end to the other, and wires have no signal-directing ability, so multiple transmissions always interfere.

So why do we call SerialPlus a bus? Because it acts like one as far as all the connected devices are concerned. They can do reads, writes, and locks just as on any ordinary bus. They do not need to know about network protocols or

routing, they just ask for data from some 64-bit address and soon the data arrive.

Of course, not all ordinary buses act alike—SerialPlus acts like a high performance split-transaction bus, where the bus is released for other uses between the data request and the arrival of those data. More primitive buses are more common, where one device holds the bus, blocking all other users, until the requested data arrive; but these "unified-transaction" buses are hopelessly inefficient and not useful where high performance is desired. Split transaction buses were the first step toward concurrence, using the bus wires as efficiently as possible. However, they were only able to scale to about 10 times the performance of primitive buses. Scaling further requires support for switches and some network-like features.

Moving from unified to split-transaction buses introduced several new problems: mutual exclusion/locks; ordering; and resource allocation. Many users encounter these issues for the first time when they make a leap from unified-transaction buses to SerialPlus or SCI, not realizing that these were already issues that had to be handled by split-transaction buses, and thus inappropriately consider SerialPlus or SCI to be complex—that is not a fair or reasonable comparison.

27.4.3 Concurrent Behavior is Essential for Scalability

Scalable architectures are ones that do not change their behavior as technology advances—they just run faster, and gain higher capacity.

As a contrasting example, a 1394 SerialBus system spends time arbitrating to decide which device will transmit next. The arbitration takes a fixed time, independent of technology (here technology means the currently possible signal bandwidth). This fixed delay interferes with scaling up in bandwidth—one has to send more data with each transmission as the bandwidth goes up, or else the system performance becomes limited by the constant arbitration delay times. Making packets longer is not a minor problem—it causes backward compatibility problems or inefficiencies, and infrastructure problems (especially for interfaces and bridges that have to provide buffer storage for packets).

Arbitration also requires a time proportional to the physical size of the connected system, which thus forces limits on the possible system size. Furthermore, when transmitting data, 1394 sends the same information over all the cables in the system. Thus adding cables does not add capacity for 1394, as it does for SerialPlus or SCI.

This is not a criticism of 1394, which was designed for absolute minimum cost in a single desktop I/O environment, consciously ignoring scalability considerations, which were in those days believed to be expensive (we were wrong). However, for high-end SerialBus users the resulting problems are already real, and gradually more and more users will bump into the built-in limits.

Now that technology has advanced to the point where the cost of logic gates is not the primary constraint in a system design, there are big advantages to using scalable architectures, which can just keep growing and adapting smoothly to use new technology as it evolves, generation after generation.

Standardized scalability is good for the consumer, but may not fit the business models of established manufacturers. Planned obsolescence and sharply defined market segments can support higher profits on a continuing basis. Standardized scalability replaces this controlled environment with free market forces, which make profits uncertain and the future unpredictable.

Thus the consumer's interests are always aligned with the underdog's—it's a very dynamic world.

References

1. Auspex. 4Front. http://www.auspex.com/pdf/tr24.pdf
2. R. Clark. *SCI Interconnect Chipset and Adapter: Building Large Scale Enterprise Servers with Pentium II Xeon SHV Nodes.* http://www.dg.com/about/html/sci_interconnect_chipset_and_a.html
3. Data General. http://www.dg.com/about/html/white_papers.html
4. Data General's NUMALiiNE Technology: The Foundation for the AV 25000. http://www.dg.com/aviion/html/av_25000_enterprise_server.html, http://www.dg.com/about/html/av25000_foundation.html, http://www.dg.com/aviion/html/av_20000_enterprise_server.html, http://www.dg.com/aviion/html/av_20000_technical_overview.html
5. Dolphin Interconnect Solutions. http://www.dolphinics.com
6. Dolphin Interconnect Solutions. Application Notes. http://www.dolphinics.com/dolphin2/interconnect/applications/apps.html
7. Interconnect Systems Solution. http://www.iss-us.com
8. Paderborn Center for Parallel Computing. http://www.upb.de/pc2
9. Lockheed Martin. RelianetSCI. http://www.lmco.com/minn/raj.htm
10. Scali. http://www.scali.com/Presentation/sld011.htm
11. Sequent. NUMA-Q 1000. http://www.sequent.com/products/servers/numaq1000/
12. Sequent. NUMA-Q 2000. http://www.sequent.com/products/highend_srv/
13. Sequent. White Papers. http://www.sequent.com/solutions/whitepapers/index.html
14. SGI. http://www.SCIzzL.com/SGIarguesForSCI.html
15. SCIzzL. http://www.SCIzzL.com
16. Siemens. HPCLINE. http://www.siemens.com/computer/hpc/en/hpcline2/index.htm, http://www.siemens.com/computer/hpc/en/hpcline5/index.htm
17. Siemens. RM600E. http://manuals.mchp.siemens.de/servers/rm/rm_us/rm_pdf/rm600e37.pdf
18. Vitesse Semiconductor. http://www.vitesse.com

List of Contributors

A. Acharya
Department of Computer Science
University of California
Santa Barbara, CA 93103
USA
acha@cs.ucsb.edu

G. Acher
Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und
Rechnerorganisation (LRR)
D-80290 München
Germany
acher@in.tum.de

G. Alléon
Aérospatiale Joint Research Centre
12 rue Pasteur, BP 76
F-92152 Suresnes Cedex
France
guillaume.alleon@siege.aerospatiale.fr

A. Belias
Rutherford Appleton Laboratory
Didcot
Oxfordshire
OX11 OQX
United Kingdom
a.belias@rl.ac.uk

T. Bemmerl
RWTH Aachen
Lehrstuhl für Betriebssysteme
Kopernikusstr. 16
D-52056 Aachen
Germany
contact@lfbs.rwth-aachen.de

A. Bogaerts
CERN
CH-1211 Geneva 23
Switzerland
andreas.johannes.bogaerts@cern.ch

D. Botterill
Rutherford Appleton Laboratory
Didcot
Oxfordshire
OX11 OQX
United Kingdom
d.botterill@rl.ac.uk

M. Brune
Konrad-Zuse-Zentrum für
Informationstechnik Berlin (ZIB)
Takustr. 7
D-14195 Berlin-Dahlem
Germany
brune@zib.de

H. Bugge
Scali AS
Hvamstubben 17
N-2013 Skjetten
Norway
hob@scali.no

R. Butenuth
Operating Systems and Distributed Systems Research Group
Universität-GH Paderborn
Fürstenallee 11
D-33095 Paderborn
Germany
butenuth@upb.de

E. Cecchet
SIRAC Lab.
INRIA Rhône-Alpes
ZIRST - 655, avenue de l'Europe
F-38330 Montbonnot Saint-Martin
France
emmanuel.cecchet@inrialpes.fr

J. Dawson
Argonne National Laboratory
Argonne, IL 60439
USA
jwd@hep.anl.gov

E. Denes
CERN
CH-1211 Geneva 23
Switzerland
ervin.denes@cern.ch

A. C. Döring
Medical University of Lübeck
Institute of Computer Engineering
Ratzeburger Allee 160
D-23538 Lübeck
Germany
doering@iti.mu-luebeck.de

M. Dormanns
RWTH Aachen
Lehrstuhl für Betriebssysteme
Kopernikusstr. 16
D-52056 Aachen
Germany
contact@lfbs.rwth-aachen.de

M. Fischer
Universität Mannheim
Institut für Technische Informatik
Rhenania-Geb. B6, 26
D-68159 Mannheim
Germany
mfischer@mufasa.informatik.uni-mannheim.de

F. Giacomini
CERN
CH-1211 Geneva 23
Switzerland
francesco.giacomini@cern.ch

D. B. Gustavson
SCIzzL/Santa Clara University
1946 Fallen Leaf Lane
Los Altos, CA 94024-7206
USA
dbg@scizzl.com

J. S. Hansen
Department of Computer Science
University of Copenhagen (DIKU)
Universitetsparken 1
DK-2100 Copenhagen East
Denmark
cyller@diku.dk

A. T. Haugsdal
Scali AS
Hvamstubben 17
N-2013 Skjetten
Norway
ath@scali.no

R. Hauser
CERN
CH-1211 Geneva 23
Switzerland
reiner.hauser@cern.ch

O. Heinz
Paderborn Center
for Parallel Computing
Universität-GH Paderborn
D-33102 Paderborn
Germany
heinz@upb.de

H.-U. Heiss
Operating Systems and Distributed
Systems Research Group
Universität-GH Paderborn
Fürstenallee 11
D-33095 Paderborn
Germany
heiss@upb.de

H. Hellwagner
Institute of Information Technology
University of Klagenfurt
A-9020 Klagenfurt
Austria
hermann.hellwagner@uni-klu.ac.at

G. Horn
SINTEF Electronics and Cybernetics
Forskningsveien 1
P.O. Box 124 Blindern
N-0314 Oslo
Norway
Geir.Horn@ecy.sintef.no

C. Hortnagl
CERN
CH-1211 Geneva 23
Switzerland
christian.hortnagl@cern.ch

R. Hughes-Jones
University of Manchester
Oxford Road
Manchester, M13 9PL
United Kingdom
r.hughes-jones@man.ac.uk

L. P. Huse
Scali AS
Hvamstubben 17
N-2013 Skjetten
Norway
lph@scali.no

M. Ibel
Department of Computer Science
University of California
Santa Barbara, CA 93103
USA
ibel@cs.ucsb.edu

W. Karl
Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und
Rechnerorganisation (LRR)
D-80290 München
Germany
karlw@in.tum.de

A. Keller
Paderborn Center
for Parallel Computing
University of Paderborn
Fürstenallee 11
D-33102 Paderborn
Germany
kel@upb.de

R. Kleber
Institut für Informatik
Technische Universität München
D-80290 München
Germany
kleber@in.tum.de

P. T. Koch
Department of Computer Science
University of Copenhagen (DIKU)
Universitetsparken 1
DK-2100 Copenhagen East
Denmark
koch@diku.dk

H. Kohmann
Dolphin Interconnect Solutions
Olaf Helsets vei 6
N-0621 Oslo
Norway
info@dolphinICS.no

S. Kolya
University of Manchester
Oxford Road
Manchester, M13 9PL
United Kingdom
scott@a3.ph.man.ac.uk

C. Kurmann
Laboratory for Computer Systems
Swiss Institute of Technology (ETH)
CH-8092 Zürich
Switzerland
kurmann@inf.ethz.ch

M. Leberecht
Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und
Rechnerorganisation (LRR)
D-80290 München
Germany
markus.leberecht@in.tum.de

M. C. Liaaen
Dolphin Interconnect Solutions
Olaf Helsets vei 6
N-0621 Oslo
info@dolphinICS.no
Norway

V. Lindenstruth
Institute for High Energy Physics
Schröder Str. 90
D-69120 Heidelberg
Germany
ti@ihep.uni-heidelberg.de

G. Lustig
Medical University of Lübeck
Institute of Computer Engineering
Ratzeburger Allee 160
D-23538 Lübeck
Germany
lustig@iti.mu-luebeck.de

E. Maehle
Medical University of Lübeck
Institute of Computer Engineering
Ratzeburger Allee 160
D-23538 Lübeck
Germany
maehle@iti.mu-luebeck.de

M. Maier-Stahel
Operating Systems and Distributed
Systems Research Group
Universität-GH Paderborn
Fürstenallee 11
D-33095 Paderborn
Germany
stahlie@upb.de

D. Mercer
University of Manchester
Oxford Road
Manchester, M13 9PL
United Kingdom
d.mercer@man.ac.uk

R. Middleton
Rutherford Appleton Laboratory
Didcot
Oxfordshire
OX11 OQX
United Kingdom
r.middleton@rl.ac.uk

W. Obelöer
Medical University of Lübeck
Institute of Computer Engineering
Ratzeburger Allee 160
D-23538 Lübeck
Germany
obeloeer@iti.mu-luebeck.de

M. Oberhuber
Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und
Rechnerorganisation (LRR)
D-80290 München
Germany
oberhube@in.tum.de

M. Ohlenroth
Fakultät für Informatik
Technische Universität
Chemnitz-Zwickau
D-09111 Chemnitz
Germany
matthias.ohlenroth@informatik.
tu-chemnitz.de

K. Omang
Scali AS
Hvamstubben 17
N-2013 Skjetten
Norway
knuto@scali.no

T. Priol
IRISA/INRIA
Campus de Beaulieu
F-35042 Rennes Cedex
France
priol@irisa.fr

A. Reinefeld
Konrad-Zuse-Zentrum für
Informationstechnik Berlin (ZIB)
Takustr. 7
D-14195 Berlin-Dahlem
Germany
ar@zib.de

C. René
IRISA/INRIA
Campus de Beaulieu
F-35042 Rennes Cedex
France
crene@irisa.fr

H. Richter
Institut für Informatik
Technische Universität München
D-80290 München
Germany
richterh@informatik.tu-muenchen.de

X. Rousset de Pina
SIRAC Lab.
INRIA Rhône-Alpes
ZIRST - 655, avenue de l'Europe
F-38330 Montbonnot Saint-Martin
France
xavier.rousset@inrialpes.fr

E. Rustad
Scali AS
Hvamstubben 17
N-2013 Skjetten
Norway
eir@scali.no

H. Ry
Scali AS
Hvamstubben 17
N-2013 Skjetten
Norway
hwr@scali.no

K. Schauser
Department of Computer Science
University of California
Santa Barbara, CA 93103
USA
schauser@cs.ucsb.edu

J. Schlereth
Argonne National Laboratory
Argonne, IL 60439
USA
jls@hep.anl.gov

M. Schmitt
Department of Computer Science
University of California
Santa Barbara, CA 93103
USA
schmittm@cs.ucsb.edu

K. Scholtyssik
RWTH Aachen
Lehrstuhl für Betriebssysteme
Kopernikusstr. 16
D-52056 Aachen
Germany
contact@lfbs.rwth-aachen.de

M. Schulz
Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und
Rechnerorganisation (LRR)
D-80290 München
Germany
schulzm@in.tum.de

J. Simon
Konrad-Zuse-Zentrum für
Informationstechnik Berlin (ZIB)
Takustr. 7
D-14195 Berlin-Dahlem
Germany
simon@zib.de

T. Stricker
Laboratory for Computer Systems
Swiss Institute of Technology (ETH)
CH-8092 Zürich
Switzerland
tomstr@inf.ethz.ch

H. Taskin
Operating Systems and Distributed
Systems Research Group
Universität-GH Paderborn
Fürstenallee 11
D-33095 Paderborn
Germany
bond@upb.de

J. Weidendorfer
Institut für Informatik
Technische Universität München
D-80290 München
Germany
josef.weidendorfer@in.tum.de

P. Werner
CERN
CH-1211 Geneva 23
Switzerland
per.werner@cern.ch

F. Wickens
Rutherford Appleton Laboratory
Didcot
Oxfordshire
OX11 OQX
United Kingdom
f.wickens@rl.ac.uk

Subject Index

Active Messages, 98, 209, 224, 240, 247, 267–277, 279, 351, 356, 357, 359, 444
address mapping, 23, 24, 26, 32, 49, 61, 71, 72, 74, 78–80, 86, 128, 177, 180, 181, 183, 184, 187, 193, 197, 199, 212–214, 219, 225, 226, 242, 263, 270, 273, 277–279, 282, 293–296, 300, 306, 315, 321, 322, 325, 327, 329
ATLAS, 31, 397, 398, 402, 403, 410, 411, 413
ATM, 50, 110, 209, 224, 228, 239, 240, 245, 247, 269, 270, 346, 402, 413, 456
ATOLL, 32
atomic transaction, 21, 201, 293, 306
ATT, 23, 74, 75, 183, 194, 198, 279, 293
Auspex, 476
AViiON, 26, 383

B-Link, 72, 73, 91–93, 95, 99, 128–130, 134, 136–138, 142, 147, 148, 167–169, 172–174, 419, 420, 427
bandwidth allocation, 8, 15, 30, 106, 110, 111, 113, 125
BB&N, 467
BIP, 62–65, 67, 241

cache coherence, 5, 9–11, 19–22, 26, 27, 29, 30, 34, 58, 71, 151, 294, 430, 465, 468, 477
call tree, 350, 354, 363
CC-NUMA, 26, 27, 31, 33, 34, 328, 329, 394, 430, 473, 474
CCS, 434, 436–438, 440
Cilk, 352, 355, 356, 358, 359, 362
CluStar, 61, 369
Cobra, 333, 340–347
CODEX, 422, 428
Compaq, 32, 33, 257
compute cluster, 3, 9, 32, 33, 40, 89, 209, 210, 222, 225, 307, 314, 323, 329, 367, 443
Concert, 356, 363
concurrent bus, 469, 478
contention, 251, 336, 343, 390, 408
Convex, 23, 26, 34, 282, 394, 473
CORBA, 264, 265, 333–341, 343–347
COTS, 257, 260, 367, 379, 469, 475
Cray, 39, 45, 46, 48, 52, 56, 57, 60, 61, 63, 65, 66, 249, 258, 269, 308, 455, 459, 475, 476

DASH, 33
data acquisition, 3, 7, 25, 31, 83, 125–127, 129, 134, 146, 147, 151, 152, 164, 413, 473
Data General, 26, 34, 114, 383, 474, 475, 477
dataflow, 349, 352–356, 362, 363
deadlock (avoidance), 18, 47, 115–117, 119, 120, 134, 203, 269, 271, 427, 433, 468
device driver, 23, 25, 65, 71, 128, 130, 179, 185, 187, 188, 207, 219, 220, 231, 232, 242, 252, 254, 264, 270, 283, 298, 309, 313, 327, 329, 330, 345
direct deposit, 39, 41, 42, 44, 52–54, 56, 58, 65–67
DMA, 25, 30, 48, 52–54, 56, 58, 61, 67, 72–74, 76, 78, 80, 82, 86, 90, 91, 93, 95, 97, 128, 191, 194–196, 202, 203, 211, 220, 222, 399, 406
Dolphin, 10, 16, 17, 23–25, 30, 31, 35, 39, 45, 48, 54, 65, 71, 72, 77, 79, 82, 83, 85, 86, 89, 100, 114, 115, 117, 127, 128, 130, 161, 167, 179, 180, 182–185, 188–190, 211, 212, 220, 221, 223, 226, 227, 234, 251, 257, 274, 277, 279, 292, 302, 309, 314, 323, 327, 330, 341, 357, 367, 369, 392, 400, 406, 413, 417, 467, 469, 470, 472–474, 476

Dolphin driver, 180, 183–185, 188–190
DSM, 4, 5, 8, 12, 22–26, 32, 34, 35, 89, 191, 192, 207, 210–215, 218–221, 225–227, 263–265, 291–293, 296, 297, 307, 308, 351, 417, 418, 422, 428, 430

error handling, 9, 10, 25, 72, 76, 130
Ethernet, 44, 50, 110, 214, 223, 231, 232, 236, 237, 239, 241, 245, 247, 335, 346, 459

Fast Ethernet, 185, 209, 234–236, 242, 369–372, 377–379, 392, 434, 456, 457
Fast Messages, 209, 225, 252, 261, 308
Fastbus, 467
FibreChannel, 469, 470, 475, 476
FireWire (IEEE 1394, SerialBus), 466
flow control, 6, 9, 12, 14–16, 32, 44, 47, 106–108, 110, 114, 160, 165, 203, 213, 226, 233, 237, 401
forward progress, 18, 125, 355
FPGA, 73, 89, 91, 93–96, 99, 100, 121, 431
Futurebus+, 465–469
FutureIO, 35, 475–478

Gigabit Ethernet, 39, 50, 51, 65, 67, 125, 413, 476
GROMOS, 383, 384, 386, 387, 392, 393, 395
GSN, 32

Harness, 247
Hewlett Packard, 23, 26, 34, 282, 383, 394, 467, 473
HIPPI, 32, 475, 476
hpcLine, 367, 476

IBM, 44, 46, 83, 114, 249, 257, 269, 383, 393, 394, 455, 459, 472, 474, 475
iCore, 470
idle symbol, 16–19, 107, 111–113, 142
IEEE 1394 (SerialBus), 35, 466, 477–479
IEEE 1596 (SCI), 3, 22, 23, 85, 191, 192, 399, 470–472
IEEE P2100, 477
iLink (IEEE 1394, SerialBus), 466
Intel, 26, 28, 32, 46, 47, 81, 90, 180, 182, 225, 226, 234, 241, 247, 269, 302, 323, 367–369, 379, 392–394, 430, 434, 471, 474, 476
Intel chip-set
– BX, 65, 234–236, 247, 368, 369
– FX, 47, 128, 223, 234, 235, 302
– GX, 369
Interconnect Systems Solutions, 115, 473

Large Hadron Collider, 31, 125, 366, 397
LazyThreads, 362
Linpack, 373, 374
Linux, 64, 65, 86, 128, 179, 180, 182, 183, 185–188, 190, 211, 221, 223, 232, 234–236, 239, 247, 282, 314, 318, 321–324, 327, 328, 330, 351, 360, 361, 370, 379, 411, 434, 444, 445
Lockheed Martin, 473
LVDS, 22, 471, 472, 475

memory barrier, 10, 273
Memory Channel, 32, 45, 209, 329
MESI, 28, 474
mezzanine connection, 83, 84, 369, 475
microcode, 29, 34, 427
MMU, 179, 182, 183, 187, 197, 279, 427, 428
MOESI, 465
monitoring, 117, 125, 344, 351, 413, 417–421, 425, 430, 431, 434, 436, 437, 440, 441
MPI, 25, 41–44, 56, 62, 63, 65, 67, 72, 86, 161–163, 165, 188, 193, 205, 209, 210, 241, 249, 251–253, 255–260, 281, 291, 333, 334, 370, 371, 373, 374, 376–378, 413, 444
MPICH, 62, 241, 371, 372, 377–379
multithreading, 305–307
MuSE, 352–358, 360–363
Myricom, 31, 39, 45, 47, 54, 64, 190
Myrinet, 31, 32, 39, 45, 48, 50–54, 58, 61–63, 65–67, 190, 210, 224, 225, 240, 241, 247, 474

network adapter, 31, 32, 52, 54, 67, 115, 226, 274, 283, 475
network interface, 32–34, 40, 43, 45–48, 50, 52, 56, 65, 114, 117, 118, 224, 270, 429, 455, 456
network topology
– Banyan, 126, 139, 147
– multi-cube, 167, 170–172, 174
– multistage, 7, 134, 135, 139, 147
– ringlet, 7, 10, 19, 139, 141–143, 145–147, 151, 152, 155–157, 159–163, 165, 169, 171–174, 274, 341, 350, 370, 371, 410, 417

– torus, 7, 84, 134, 167, 170, 174, 258, 259, 370, 378, 438, 443, 451, 455, 457
NGIO, 35, 476–478
Norsk Data, 467, 472
NUMA, 5, 23, 31, 42, 45, 89, 136, 137, 282, 283, 286, 288, 289, 308, 328, 349, 366, 383, 386, 394, 395, 417, 418, 433
NUMA-Q, 26–29, 282, 383, 474
NUMALiiNE, 383

Origin 2000, 33, 258–260, 282, 383, 394

PAPERS, 32
ParaStation, 32, 225, 241
PCI, 25, 31, 39, 45–48, 51, 52, 54, 59, 61, 65–67, 71, 73–75, 78–83, 85, 89–92, 94, 95, 97, 100, 132, 161, 162, 165, 167, 168, 174, 182, 213, 221, 223, 241, 251, 277, 293, 294, 300, 369, 401, 403, 408, 410, 417, 418, 427, 431, 476
– PCI32, 71, 75–79, 81–83
– PCI64, 71, 75, 78, 79, 82, 84
performance evaluation, 48, 63, 89, 249, 256, 260, 418–420, 430
PIO, 194, 211, 237, 401
Pthreads, 305, 307
PULC, 241
PVM, 41–43, 56, 63, 66, 72, 86, 209, 239–241, 243, 245, 257, 281, 291, 334, 370, 459
PVM-SCI, 239–243, 245–247

queue allocation, 8, 12, 15, 16, 108–110, 113

Reliant Unix, 211, 221
ring buffer, 79, 189, 218, 233, 237, 245, 246, 254, 406, 420
RMA, 24, 29, 30, 33, 71, 73, 179, 212, 263, 267, 268, 273, 292, 322, 323, 388, 390, 393, 420, 431
RPC, 42, 189, 413
rule-based routing, 120

S3.mp, 34
SAN (system area network), 3, 6, 9, 23, 39, 47, 50, 78, 81–83, 209, 210, 241, 245, 294
SBus, 23, 24, 71, 73–77, 79–81, 83, 85, 211, 221, 222, 227, 357, 475
Scali, 25, 63, 86, 167–169, 174, 180, 182–185, 188–190, 242, 249, 251–254, 256, 257, 260, 367, 369, 380, 476
Scali driver, 180, 182–185, 188–190
ScaMPI, 63, 161, 247, 249, 253–260, 377, 379
SCI extensions, 3, 22, 23, 35, 116, 151, 293, 477
SCI packet, 9, 10, 12–14, 16–18, 73–77, 79, 91, 93, 95, 105, 106, 114, 128, 152, 168, 170, 171, 196, 202, 209, 215, 222, 225, 400, 408, 418, 419, 421
– format, 15, 107, 168
– request echo, 15, 18, 108
– request send, 15, 16, 18
– request/response echo, 14, 106, 108, 113, 114, 154, 158, 160, 162
– request/response send, 12, 16, 18, 106, 108, 113
– response echo, 15
– response send, 15, 16
SCI switch, 117, 134–137, 141, 143, 144, 147, 341, 407, 413, 444
SCI-VM, 292–300, 302, 304–309
SCILAN, 226, 235
SCINET, 125–127, 145, 148
SciOS, 307, 313, 314, 317, 324–330
SCIzzL, 468, 471
scrubber node, 19, 113
Sequent, 26, 27, 34, 115, 282, 383, 467, 472, 474, 477
Serial Express (IEEE P2100), 35, 477
SerialBus (IEEE 1394), 35, 466, 477–479
SerialPlus (IEEE P2100), 23, 35, 477–479
ServerNet, 33
SGI, 30, 32, 45, 249, 257, 282, 304, 383, 384, 393, 394, 476
SHRIMP, 32, 33, 210, 225, 430
Siemens, 25, 30, 31, 71, 83, 85, 221, 368, 369, 476
SISAL, 352, 361
SISCI, 86, 181, 305, 307, 309, 399, 400, 413, 414
SLDRAM, 23, 471
SMI, 188, 282–284, 286–289, 383, 386, 388, 389
SMiLE, 350–352, 356, 357, 359, 360, 417–419, 425–428, 430, 431
SMP, 5, 26–30, 32, 51, 185, 188, 249, 251, 253, 264, 291, 292, 296, 298, 305, 308, 309, 329, 334, 433
sockets, 40, 44, 209–216, 218, 219, 222–227, 243

Solaris, 63, 86, 185, 186, 211, 221, 234–236, 239, 282, 357, 370, 379, 434, 444, 445
split transaction, 72, 73, 90, 479
Split-C, 253, 267–269, 271–274, 276, 277, 279, 308
SSLib, 210–214, 216, 218–227, 246
store barrier, 78, 81, 220
SUN, 24, 34, 71, 77, 249, 257, 318, 379, 383, 434, 476
SyncLink, 23, 35

TCP, 39–41, 44, 64–67, 185, 209–211, 214, 215, 218, 219, 223–226, 231–233, 235, 237, 239–241, 243–245, 247, 371, 411, 413, 434, 436, 438
TreadMarks, 307, 329

U-Net, 33, 209, 210, 224, 240
UDP, 44, 209, 212, 214, 215, 219, 224, 239, 240, 243, 411, 413, 438
UMA, 136, 137, 394

VI Architecture, 33, 72, 83, 86, 205, 210, 224, 226, 241
virtual file system, 318, 329
Vitesse, 29, 115, 470, 472–474
VMMC, 32, 225

Windows NT, 65, 83, 86, 128, 185, 186, 226, 235, 239, 282, 284, 292, 295, 298, 302, 307, 309, 313–315, 317–322, 329, 370, 379, 392, 411, 413, 444
work stealing, 265, 352, 355, 356, 358, 359, 362

Yasmin, 188, 370

zero-memory-copy, 32, 41, 42, 52, 61, 62, 64, 241, 357